Software Engineering Institute

Evaluating a Service-Oriented Architecture

Phil Bianco, Software Engineering Institute
Rick Kotermanski, Summa Technologies
Paulo Merson, Software Engineering Institute

September 2007

TECHNICAL REPORT
CMU/SEI-2007-TR-015
ESC-TR-2007-015

Software Architecture Technology Initiative

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Acknowledgements

We want to thank the following people for their thoughtful feedback and discussion that greatly improved the quality of this report: Felix Bachmann, Sri Bala, Eric Hohman, Rick Kazman, Mark Klein, Richard Koch, Grace Lewis, Ed Morris, Linda Northrop, Scott Parker, and Soumya Simanta.

Abstract

The emergence of service-oriented architecture (SOA) as an approach for integrating applications that expose services presents many new challenges to organizations resulting in significant risks to their business. Particularly important among those risks are failures to effectively address quality attribute requirements such as performance, availability, security, and modifiability. Because the risk and impact of SOA are distributed and pervasive across applications, it is critical to perform an architecture evaluation early in the software life cycle. This report contains technical information about SOA design considerations and tradeoffs that can help the architecture evaluator to identify and mitigate risks in a timely and effective manner. The report provides an overview of SOA, outlines key architecture approaches and their effect on quality attributes, establishes an organized collection of design-related questions that an architecture evaluator may use to analyze the ability of the architecture to meet quality requirements, and provides a brief sample evaluation.

1 Introduction

Service-oriented architecture (SOA) is a very popular architecture paradigm for designing and developing distributed systems. SOA solutions have been created to satisfy business goals that include easy and flexible integration with legacy systems, streamlined business processes, reduced costs, innovative service to customers, and agile adaptation and reaction to opportunities and competitive threats.

One of the most valuable software engineering principles is to introduce inspection points into the software life cycle. Software architecture evaluation is a particularly important inspection point, because architecture is the bridge between business goals and the software system. Choosing and designing an architecture that satisfies functional as well as quality attribute requirements (e.g., availability, security, and performance) is vital to the success of the system. Architectural decisions have a deep and broad effect on downstream development stages. Early evaluation of the requirements and the architecture saves time and money, because fixing defects once the code is fielded is at least three times more costly [McConnell 2001].

The goal of this report is to offer practical information to assist the architecture evaluation of a system that uses the SOA approach. We provide guidance on important aspects of the architecture evaluation activity, such as enlisting stakeholders, describing the architecture, identifying architectural approaches, and probing the architects with questions about the architecture. The system-specific business goals and requirements dictate how the architecture will be probed by the evaluation team and determine the types of questions that should be asked.

This report does not introduce a new method for architecture evaluation. The Architecture Tradeoff Analysis Method® (ATAM®)—developed by the Carnegie Mellon® Software Engineering Institute (SEI)—is used as the basis for defining the activities and information that are important for an architecture evaluation of a system that uses an SOA approach. While we use the ATAM, we believe the information provided will be useful regardless of the evaluation method employed.

In SOA solutions there are *service providers*—elements offering services to be used by others—and *service users*—elements that invoke services provided by others. These categories are not mutually exclusive. A service provider may use other services, and a service user may provide a service interface. This report is targeted at the evaluation of the software architecture at the level where it describes the integration of these elements through services. This evaluation at the service integration level does not replace the need to evaluate the internal design of each service user and provider.

In this report, we do not address the evaluation of business strategy alignment, risk management, cost benefit analysis of technology adoption, product and vendor selection, or skill development.

® Architecture Tradeoff Analysis Method, ATAM, and Carnegie Mellon are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

We focus on the technical considerations of evaluating the architecture of a specific system that uses the SOA approach.

1.1 AUDIENCE FOR THIS REPORT

The report is aimed at software architects using the SOA approach and anyone concerned with evaluating SOA solutions. The technical discussion presumes some familiarity with Web services technology and distributed software development.

The report should be particularly useful to an architecture evaluation team evaluating an SOA-based architecture using the ATAM or a similar method. It will help them define an appropriate group of stakeholders, formulate important quality attribute requirements, identify architectural approaches used in the solution, understand how those approaches affect system qualities, and probe the architecture about SOA-specific design concerns. The report can also guide SOA architectural choices made by software architects in the planning and designing phases.

1.2 STRUCTURE OF THIS REPORT

Section 2 of the report defines SOA and Web services from the point of view of software developers. Section 3 discusses key aspects of any architecture evaluation: selecting stakeholders to participate in the evaluation, specifying quality attribute requirements that are important in SOA, and describing the architecture of an SOA-based system. Sections 4 and 5 are the core of the report. Section 4 describes architectural approaches for SOA-based systems and their tradeoffs. Section 5 provides a list of design questions that influence quality attributes and can be used to probe the architecture during the evaluation. This section also contains references to general quality attribute scenarios described in Appendix A. Section 6 is a sample application of the guidelines in Sections 3, 4, and 5. Section 7 has our concluding remarks.

The report contains many technical terms and acronyms used by the SOA community. We provide a glossary of technical terms and acronyms in Appendix B to avoid extensive explanations in the report text. We provide a more extensive acronym list in Appendix C that includes both technical and SEI-specific acronyms used in the report.

2 What Is Service-Oriented Architecture?

There are many definitions of SOA but none are universally accepted. What is central to all, however, is the notion of *service*. For an SOA, a service

- is self-contained. The service is highly modular and can be independently deployed.
- is a distributed component.[1] The service is available over the network and accessible through a name or locator other than the absolute network address.
- has a published interface. Users of the service only need to see the interface and can be oblivious to implementation details.
- stresses interoperability. Service users and providers can use different implementation languages and platforms.
- is discoverable. A special directory service allows the service to be registered, so users can look it up.
- is dynamically bound. A service user does not need to have the service implementation available at build time; the service is located and bound at runtime.

These characteristics describe an ideal service. In reality, services implemented in service-oriented systems lack or relax some of these characteristics, such as being discoverable and dynamically bound.

We define SOA as *an architectural style where systems consist of service users and service providers*. An architectural style defines a vocabulary of component and connector types, and constraints on how they can be combined [Shaw 1996]. For SOA, the basic component types are service user and service provider. Auxiliary component types, such as the enterprise service bus (ESB) and the directory of services, can be used. SOA connector types include synchronous and asynchronous calls using SOAP, bare http, and messaging infrastructure. Many properties can be assigned to these component and connector types, but they are usually specific to each implementation technology. For example, as we will see in Section 5.1, the property "style" for messages in the Web services technology can be either "RPC" or "document." Some of the constraints that apply to the SOA architectural style are

- Service users send requests to service providers.
- A service provider can also be a service user.
- A service user can dynamically discover service providers in a directory of services.
- An ESB can mediate the interaction between service users and service providers.

[1] In practice a service implementation may consist of a collection of components. They work together to deliver the functionality the service represents.

2.1 SOA AND WEB SERVICES

Although much has been written about SOA and Web services, there still is some confusion between these two terms among software developers. SOA is an architectural style, whereas Web services is a technology that can be used to implement SOAs. The Web services technology consists of several published standards, the most important ones being SOAP and WSDL. Other technologies may also be considered technologies for implementing SOA, such as CORBA. Although no current technologies entirely fulfill the vision and goals of SOA as defined by most authors, they are still referred to as SOA technologies. The relationship between SOA and SOA technologies is represented in Figure 1. Much of the technical information in this report is related to the Web services technology, because it is commonly used in today's SOA implementations.

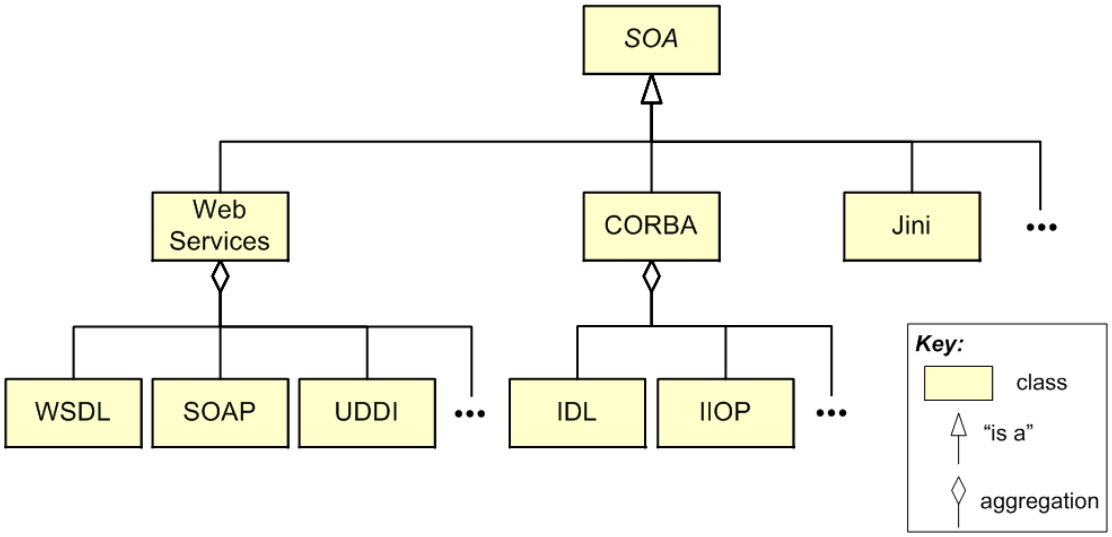

Figure 1: SOA and SOA Technologies

2.2 DRIVERS FOR SOA

In large organizations, the following types of organizational, business, and technology changes drive a desire to reap the benefits of SOA:

- integration with legacy systems
- corporate mergers
- realignment of responsibilities through business reorganizations
- changing business partnerships (e.g., relationships with suppliers and customers)
- modernization of obsolete systems for financial, functional, or technical reasons
- acquisition or decommission of software products
- sociopolitical forces related to or independent of the drivers cited above

These forces lead to SOA because they involve significant application integration efforts. When an integrated application changes, risky and costly modifications to other applications are frequently required. As system interconnections become more pervasive and the pace of business and process changes increases, the inability to integrate efficiently can cause the failure of a business. SOA is seemingly ideal for these situations.

Distributed systems technologies of the past, such as CORBA, didn't achieve broad adoption in part, because standards were not widely endorsed by CORBA vendors. More recent SOA technologies, such as Web services, seem to be off to a better start as they begin to be widely adopted. One possible explanation for the change in attitude is that the need to interoperate with applications outside the scope of a given organization is becoming more vital. The notion of software as a service (SaaS) delivery is intended to allow organizations to selectively purchase, mix, match, and change sources of services to their business advantage. The goal of the service-oriented approach is to enable the composition of new or existing services and applications in a technologically heterogeneous environment. However, many of the issues and concerns encountered and addressed in distributed systems designs over the past 20 to 30 years also apply directly to SOA. Significant shortcomings of integration approaches are related to the independent entropy (or movement to disorder) of connected but separately managed applications. Since many technical issues remain, a careful evaluation of a system's design decisions is important to ensure that an SOA solution can attain the benefits advertised by proponents of the SOA approach.

3 Stakeholders, Quality Attributes, and Architecture Representation for SOA

In any software architecture evaluation, three activities are critical to success: (1) selecting a representative constituency of stakeholders to provide input in the evaluation; (2) specifying the quality attribute requirements that derive from business goals in a precise way; and (3) describing the architecture in an expressive and comprehensive way. This section discusses how these activities should be carried out when the architecture in question is SOA based.

3.1 STAKEHOLDERS

The architecture evaluation should allow participants to express concerns and see how their concerns are addressed. A broader constituency of stakeholders decreases the risk of overlooking important architectural concerns. One of the challenges of eliciting quality attribute requirements for a system is that it may not be possible to know all the stakeholders. This is especially true in SOA-based systems that provide public services and/or search for services at runtime. Most of the roles listed below are common to all systems; there are some roles that are unique to an SOA-based system (these are italicized). The specific stakeholders chosen for an evaluation will depend on the needs of the organization. At a minimum, the following stakeholders should be invited to participate in the architecture evaluation of a system:

System Producers

- **software architects.** Their main responsibilities include experimenting with and deciding between different architectural approaches, creating interface and component specifications, and validating the architecture against the functional and quality attribute requirements. The architects create documentation that articulates the architectural vision to other stakeholders, documenting the risks and tradeoffs of the architecture as well as the rationales for design decisions. Architects also ensure that the implementation conforms to the architecture.
- **developers.** Their main responsibilities include implementing the architectural elements of the system according to the architecture specification, offering expertise during detailed design processes, and conducting experiments or creating prototypes to validate an architectural approach.
- ***service usage regulators.*** Their main responsibilities include creating policies for service usage, such as specifying that services must conform to certain standards, and possibly placing constraints on the services that can be used (e.g., specifying trusted sources for services). Another responsibility might be crafting service level agreements between organizations.
- **testers.** Their main responsibilities include planning tests of the systems, executing all planned tests, recording the results of all planned tests, and reporting defects.

- **integrators.** Their main responsibilities are to ensure that the architecture and implementation conform to open and widely accepted standards, and to advocate architectural approaches that simplify service integration, upgrades, and replacements.
- **maintenance developers.** Their main responsibilities include modifying the software to correct defects and adapting the software when environmental changes occur (e.g., hardware or operating system changes).
- **project managers.** Their main responsibilities include managing the development effort, creating the project plan, and tracking the progress of the project.
- **chief information officers (CIOs).** The CIO works with the architects, business analysts, and developers to ensure that a solution will integrate well with existing systems, applications, and infrastructure.

System Consumers

- **chief security officers (CSOs).** The CSO works with the architects, business analysts, and developers to ensure that all information security policies are followed.
- **business managers.** Their primary interest is to ensure that the application supports the organization's business goals and that the architects understand all legal and regulatory implications.
- **business analysts/customers.** Their primary interests and responsibilities are to acquire and transmit to developers the knowledge of the business domain and functional and quality attribute requirements of the system.
- **end users.** Their main responsibilities include learning to operate the system, preparing and entering inputs, and interpreting the output from the system. They also generate system requirements.
- ***developers of service users.*** If the system offers services to external service user applications, the architects or developers who are responsible for these external clients should also be invited. These external developers may provide input on application program interface (API) design and desired quality of service (e.g., availability).
- **maintenance developers.** They are responsible for general maintenance duties (described above) with the subtle difference that they would most likely not be able to modify services and would often be forced to modify other parts of the system. The inability to modify services would be similar to buying off-the-shelf software.

Infrastructure Providers

- **system administrators.** Their responsibilities include attaining a good understanding of the system operation for troubleshooting problems that arise during and after deployment. They usually assume most duties associated with computer security in an organization (i.e., upkeep of firewall and intrusion detection systems, management of access rights, and applying patches to software and operating systems).
- **network administrators.** The network administrator maintains the network infrastructure and troubleshoots problems with routers, switches, and computers on the network.

- **database administrators.** They create and maintain databases, ensuring data integrity and consistent performance of the database management systems.
- ***external developers of service providers.*** If the system is going to access external services, the architects or developers who are responsible for those external services should also participate in the architecture evaluation. They may contribute requirements for interaction with their services, as well as knowledge of qualities and limitations of their systems.

3.2 QUALITY ATTRIBUTE REQUIREMENTS

A quality attribute is a property of a system by which some stakeholders will judge its quality. Quality attribute requirements, such as those for performance, security, modifiability, reliability, and usability, have a significant influence on the software architecture of a system [SEI 2007]. Quality attribute requirements can be specified using quality attribute scenarios. Appendix A provides several examples of scenarios that are common in SOA systems.

The use of a service-oriented approach positively impacts some quality attributes, while introducing challenges for others. This section summarizes the effect of SOA on different quality attributes. O'Brien and colleagues provide a more detailed analysis [O'Brien 2005].

Improved interoperability is one of the most prominent benefits of SOA. With Web services technology, for example, service users can transparently call services implemented in disparate platforms using different languages. In this technology, the goal of syntactic interoperability[2] is supported by two basic standards: WSDL and SOAP. There are also additional standards such as UDDI, BPEL, and WS-Security that provide other capabilities to systems developed with Web services technology. However, not all Web services platforms implement the same version of these additional standards, and in practice, interoperability is not as easy to achieve as is advertised.

Modifiability is the ability to make changes to a system quickly and cost-effectively. SOA promotes a looser coupling between service users and providers. Services are modular and self-contained, reducing the number of usage dependencies between service users and providers. Therefore the cost of modifying these services is reduced. Changing the interface of a published service is still a challenge, but SOA solves this problem through versioning mechanisms and more flexible contracts specified in Extensible Markup Language (XML). Modifiability requirements that involve finding and incorporating a new service are also easier in SOA.

Performance in an SOA context is usually measured by average case response times or throughput. In most cases, SOA performance is negatively impacted because

- SOA enables distributed computing, and the need to communicate over a network increases the response time.

[2] Web Services provide primarily syntactic interoperability. Whether two components can interoperate also depends on their semantic agreement about the meaning of data and operations. Throughout this report, interoperability discussions refer primarily to syntactic interoperability.

- Intermediaries, such as the directory of services and proxies that perform data marshalling, cause some performance overhead.
- Standard messaging formats (e.g., XML) increase the size of messages and hence the time to process requests.

Security is also a challenge in SOA, especially when external services or public directories of services are used. A common problem is the negative impact of vendor-specific security features on interoperability. The WS-I organization has recently published the Basic Security Profile Version 1.0 to try to remedy this problem [WS-I 2007]. Other security design concerns are covered in Sections 5.5, 5.6, and 5.7.

Testability is the degree to which a system facilitates establishing test criteria and performing tests. Testing a system that uses SOA is more complex. First, it is more difficult to set up and trace the execution of a test when system elements are deployed on different machines across a network. Second, the source code of external services is often unavailable, so test cases must be defined based on published interfaces. If the source code were available, the tester would be able to ensure better code coverage. Also, in Web services solutions, sometimes the error is in an XML document (e.g., a WSDL definition), and dealing with raw XML is cumbersome. Finally, in cases where services are discovered at runtime, it may be impossible to determine which service is being used until the service is executing. Because SOA involves distributed components in a heterogeneous environment and may require distributed transactions, achieving high reliability is also challenging.

3.3 ARCHITECTURE DESCRIPTION OF AN SOA

The architecture description is required to perform an architecture evaluation. In the architecture description, multiple views are necessary to communicate the architecture to the various stakeholders [Clements 2002a]. Module views show the structure of units of implementation; Runtime views (also known as Component & Connector views) show the components that have runtime presence; Deployment views show the hardware infrastructure and deployment artifacts; and the Data Model view shows the organization of entities in a database. There are other types of views, and the architect should choose which views to document and which views to document in detail, based on the stakeholders' needs and the kinds of structures found in the system.

But which view best shows the service-oriented aspect of an architecture? In Section 2, we said that a service is a distributed component whose implementation details can be hidden. The distributed nature of a service and the interaction between a service user and service provider are manifested at runtime. Thus, the Runtime view best captures a service-oriented design. Using the terminology of the Views & Beyond approach for software architecture documentation [Clements 2002a], we can say that SOA is a style of the Component & Connector view type.

SOA solutions can be very rich, comprising external services and special components, such as the ESB. It is usually beneficial to complement the structural diagrams in Runtime views with behav-

ior diagrams (e.g., UML sequence diagrams) that describe the sequence of interactions occurring in specific transactions.

Section 6.2.2 provides a small sample architecture (including a Runtime view diagram and a sequence diagram) that shows the behavior of the system when a specific stimulus arrives.

Although the architecture description focuses on the structures of the system being implemented, other types of documentation are also important. Understanding and documenting the business process flow is very important in SOA solutions. Section 4.3 has more information about process specification, automation, and orchestration.

4 SOA Architectural Approaches

In an architecture evaluation, we often don't have time to look at all the architectural elements of the system. Architecture evaluators understand how different architectural approaches and patterns affect quality attributes. Therefore, to evaluate whether a software architecture can meet the quality attribute requirements, we can focus on the architectural approaches used in the system. For the evaluation of an SOA system, we focus primarily on service integration and communication patterns, rather than the architectures of the underlying integrated applications.

Beyond traditional distributed systems design concerns—such as network communication, security, transaction management, naming, and location—which architectural approaches are different with SOA? This section describes common and emerging SOA architectural approaches that will be factors in evaluating an SOA.

4.1 SOA COMMUNICATION APPROACHES

Each interaction between a service user and a service provider in an SOA can be implemented differently. The implementation alternatives impact important quality attributes of the system, such as interoperability and modifiability. In a pure Web services solution, the SOAP protocol is used. However, the architect can also avoid SOAP and use a simpler approach, such as Representational State Transfer (REST). Another option is to use messaging systems, such as Microsoft MSMQ and IBM Websphere MQ (previously called MQSeries). These alternatives and related quality attribute concerns are discussed below. An SOA environment may involve a mix of these alternatives along with legacy and proprietary communication protocols, such as IIOP, DCOM, DCE, and SNA/LU6.2.

4.1.1 SOAP-Based Web Services

Web services is a technology commonly used to implement SOAs. Service interfaces are defined in the WSDL language, and service users and service providers communicate using the SOAP protocol. Two attributes in a WSDL interface, "style" and "use," define the SOAP communication between service users and providers. The style attribute has two possible values: "RPC" and "document." The use attribute refers to data encoding and has two possible values: "encoded" or "literal." Consequently there are four possible combinations of these two attributes. Two combined options that are common in practice are RPC-encoded and document-literal. They are described next.

RPC-Encoded SOAP

In the RPC style, the SOAP message is equivalent to an XML-based remote method call. The name and type of each argument is part of the WSDL interface definition. The body of the SOAP request necessarily contains an element indicating the operation name and sub-elements corresponding to the operation arguments. The *encoded* attribute indicates that data is serialized

using a standard encoding format. This format is defined by the SOAP specifications and contains rules to encode primitive data types, strings, and arrays. Figure 2 represents an RPC-encoded interaction.

The RPC-encoded style was popular in the first years of the Web services technology because of its simple programming model and the similarity between service operations and object methods. However, it is not a good choice, because interoperability problems can arise from deficiencies in the SOAP-encoding specifications [Ewald 2002].

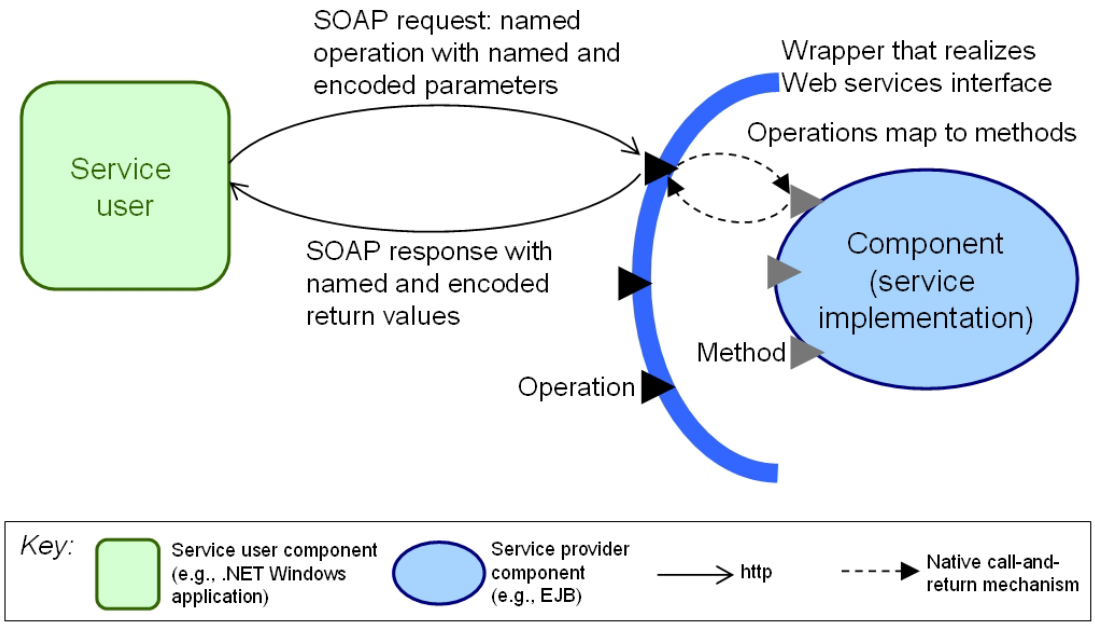

Figure 2: RPC-Encoded Interaction

Document-Literal SOAP

The SOAP message body in a document-literal style request can contain arbitrary XML (the business document). The WSDL definition does not have to specify named parameters, and the XML content of the message body does not follow a standard structure as in the RPC style. The literal attribute indicates that no standard encoding format is used—data in the SOAP body is formatted and interpreted using the rules specified in XML schemas created by the service developer. The XML schemas that define the data structure of the request and the response are the key elements in the interface definition. Figure 3 shows a document-literal interaction.

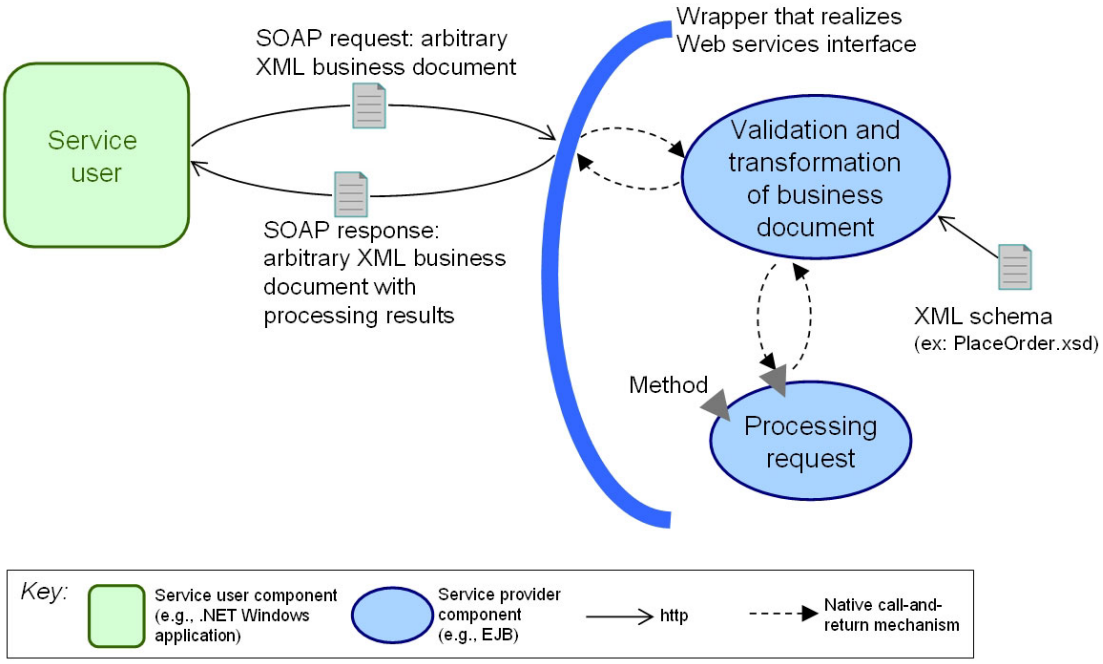

Figure 3: Document-Literal Interaction

Table 1 compares the RPC-encoded and document-literal approaches with respect to different quality attributes, clarifying why document-literal is currently the most common approach for SOAP messages. The document-literal approach is recommended by the WS-I organization. In an architecture evaluation, the architect should be aware of the differences between these styles. Some Web services toolkits still use RPC-encoded as the default style; therefore, it is important that developers know how to specify the desired style when creating services.

Table 1: Comparison of RPC-Encoded and Document-LiteralApproaches

Quality Attribute	RPC-Encoded	Document-Literal
Interoperability	☹ Is less interoperable due to incompatibility in SOAP encoding across platforms	☺ Is more interoperable and recommended by WS-I
Performance	☹ May yield worse performance due to processing overhead required to encode payloads	☺ Requires no encoding overhead
	☹ Requires DOM parsing	☺ Allows other parsing technologies (e.g., SAX)
Modifiability	☺ In theory yields better modifiability because service interfaces are closer to programming language interfaces with operations and parameters. This similarity also enables the use of automatic object-to-WSDL translation.	☹ Is usually harder to implement because the XML schema definitions and the code to process and transform the XML documents are usually not created automatically
	☹ In practice, any change to the syntax of an operation requires changes in the service users, resulting in increased coupling.	☺ Yields less coupling. There is more flexibility to change the business document without affecting all service users.

4.1.2 REST

REST was proposed by Roy Fielding [Fielding 2000]. It avoids the complexity and processing overhead of the Web services protocols by using bare http. As an example, consider a weather forecast service that is publicly available and is provided by http://www.weather.com. One important REST concept is a *resource*, which is a piece of information that has a unique identifier (e.g., a uniform resource identifier (URI)). For the weather service, examples of resources include

- current weather for zip code 15213
- weather forecast for tomorrow for the city of Pittsburgh
- 10-day weather forecast for zip code 15213
- temperature averages for the city of Pittsburgh in October

In this example, there are three types of resources: current weather, weather forecast, and temperature averages. We can structure the URIs of the resources based on these three types. Parameters can be represented by elements in the URI hierarchical path or [key]=[value] pairs. The URIs corresponding to the resources we listed above could be

- http://www.weather.com/current/zip/15213
- http://www.weather.com/forecast/tomorrow/city/Pittsburgh
- http://www.weather.com/forecast/tenday/zip/15213
- http://www.weather.com/avg/city/Pittsburgh?month=10

It is no coincidence that these URIs look like what we type in a Web browser. REST relies on the http protocol for the interaction between service users and providers. The http protocol has four basic operations: POST, GET, PUT, and DELETE. In a REST design, the application of these operations to resource URIs correspond to create, retrieve, update, and delete (CRUD) operations commonly used in information systems. Thus, if the service user sends a POST request on http://www.weather.com/current/zip/15213, it is asking the service provider to create the data for the current weather in zip code 15213 using the data passed along with the request. A GET request on the same URI tells the service provider to retrieve the data for the current weather in zip code 15213 and return it in the response. A PUT request indicates that the service provider should replace the data it has with the data sent in the request. A DELETE request indicates that the service user wants the service provider to delete the data. The http protocol also defines the status codes that can be returned: 200 for OK, 201 for created, 401 for unauthorized, and so forth.

A unique characteristic of REST is that it prescribes a uniform interface—the service is exposed as information resources upon which a fixed set of operations can be applied, rather than a set of methods with different parameters. In a REST solution, for each resource we have to define a *representation*. In most cases, basic XML is the format used. Also, REST services are necessarily stateless—they don't store the conversational state across multiple requests from the same service user.

REST advocates claim several benefits over SOAP-based Web services :

- REST results in improved modifiability. For a service user to interact with a non-REST Web Service, the service user has to understand the specifics of the data contract (i.e., how data is structured) and the interface contract (i.e., what operations can be performed). Because of the uniform interface, to invoke a REST service, the service user only has to understand the data contract, because the interface contract is uniform for all services [Vinoski 2007].
- REST is easy to implement and yields high interoperability, since it only requires standard http support from both the service user and provider. It doesn't require SOAP toolkits to implement the code or an application server that supports Web services .
- REST has better performance due to its ability to cache the responses (when applicable) and to the absence of the intermediaries, message wrapping, and serialization that are required by Web services .

Web services and REST represent different paradigms to implement SOA. One is centered on the operations to be executed by the service provider. The other is focused on access to resources. In the architecture evaluation of an SOA system, the evaluation team can question which approach would be more appropriate for each service. REST is a good option for accessing static or nearly static resources. It is also useful for portable devices with limited bandwidth, because REST messages are less verbose than SOAP messages. The Web services technology offers better support for security, reliable messaging, and transaction management [MacVittie 2006]. As a result of widespread adoption, plenty of knowledge on Web services is provided on the Web and in the professional community. There is also better tool support for developing Web services, although APIs for easy development of REST solutions are being created, such as the Java API for RESTful Web services [Sun 2007b]. If the application is going to provide services to multiple users and

business partners, an alternative is to build both SOAP and REST interfaces for the same services like Amazon.com and eBay do.

4.1.3 Messaging Solutions

The interaction between service users and service providers can also be based on messaging systems, such as IBM WebSphere MQ, Microsoft MSMQ, Oracle AQ, and SonicMQ. These products offer primarily asynchronous message exchanges between distributed applications in a point-to-point (sender-receiver) or publish-subscribe fashion. Basically, the messaging system allows an administrator to configure message queues. Applications can then connect to these queues to send or receive messages, while the messaging system coordinates the sending and receiving of messages. These solutions can also be designated as event-driven architecture (EDA), in which case the messages are events and queues are often called channels.

The main benefits of messaging solutions are

- They offer great reliability with guaranteed delivery of messages.
- They promote loose coupling between message producers and consumers, and the reuse of message consumers.
- They are particularly useful when connecting disparate systems and legacy applications.
- Commercial implementations provide high scalability to support an increasing number of clients by adding more instances of message consumers.
- They may be designed to work offline (i.e., disconnected from the network). Messages are queued and stored on the sender, and when connectivity resumes, they are sent to the receiver in the same way that a PDA synchronizes with a server.

There are three main challenges in messaging systems. The first challenge is that the asynchronous programming model is more complex, particularly when the interaction is synchronous and a callback mechanism must be used (see Section 5.2). The second challenge is the performance cost to wrap data in message packets and to store (sometimes on disk) the messages on the sender and/or receiver computer. The third challenge is interoperability. Proprietary messaging systems are usually not available on all platforms. For example, Microsoft MSMQ is a Windows-only product. Moreover, messaging systems usually rely on proprietary protocols and require third-party bridges to interact with other messaging systems.

There are isolated solutions that use SOAP over messaging systems [Shah 2006, Kiss 2004], but the most important ongoing efforts today to allow messaging systems to benefit from SOAP interoperability are the WS-Reliability [OASIS 2004a] and WS-ReliableMessaging [OASIS 2006b] standards. They have much more in common than the name, as indicated in Table 2. Both standards define SOAP-based reliable messaging via acknowledgments. Vendors of Web services platforms, such as Microsoft, IBM, Sun Microsystems, and BEA, have announced support for either or both standards. The implementations of the standards often build on an existing messaging system. Both standards allow message producers and consumers implemented in different languages and on different platforms to interoperate seamlessly using the SOAP protocol. Nonetheless, the fact that there are competing specifications may itself become an obstacle to interop-

erability, though the industry seems to be moving towards WS-ReliableMessaging. One indicator of this is its prescription in the recently published WS-I Reliable Secure Profile Version 1.0.

In the architecture evaluation, if both reliability and interoperability are strong requirements, the use of products compatible with WS-ReliableMessaging is a step in the right direction.

Table 2: WS-Reliability and WS-ReliableMessaging—Who Is Who

Standard name	Web Services Reliability	Web Services Reliable Messaging
Abbreviated name	WS-Reliability	WS-ReliableMessaging
Acronyms commonly used	WSRM, WS-R	WS-RM
Standard body	OASIS	OASIS
Committee name	Web Services Reliable Messaging Technical Committee	
Current version and status	OASIS published standard V1.1, November 15, 2004	Committee Draft 04, August 11, 2006
Original champions	Sun Microsystems, Fujitsu, Hitachi, NEC, Oracle, Sonic Software	BEA, IBM, Microsoft, Tibco

4.2 INTEGRATION APPROACH – DIRECT POINT-TO-POINT VERSUS ESB

The establishment of system integration patterns and strategies for an SOA system has a significant and long-lasting impact. The two significant options for a primary integration pattern are (1) direct point-to-point and (2) hub-and-spoke. In the direct point-to-point approach, each connection between applications (i.e., each service user-provider interaction) is individually designed and cooperatively implemented, deployed, and administered. Responsibility for connectivity issues such as location, naming, security, auditing, and versioning of services is distributed among the applications.

In the hub-and-spoke approach, the interaction between service users and providers is mediated by brokering software. In the SOA space, this brokering software is usually called the ESB. The more classical term is enterprise application integration (EAI) software. Each application is designed to interact with the ESB, allowing it to manage the routing and transformation of messages between applications. Figure 2 provides a simplified comparison of ESB and point-to-point integration topologies. It is common in large organizations to have a mixture of approaches that depend on a variety of factors, such as application age and purpose of integration connectivity.

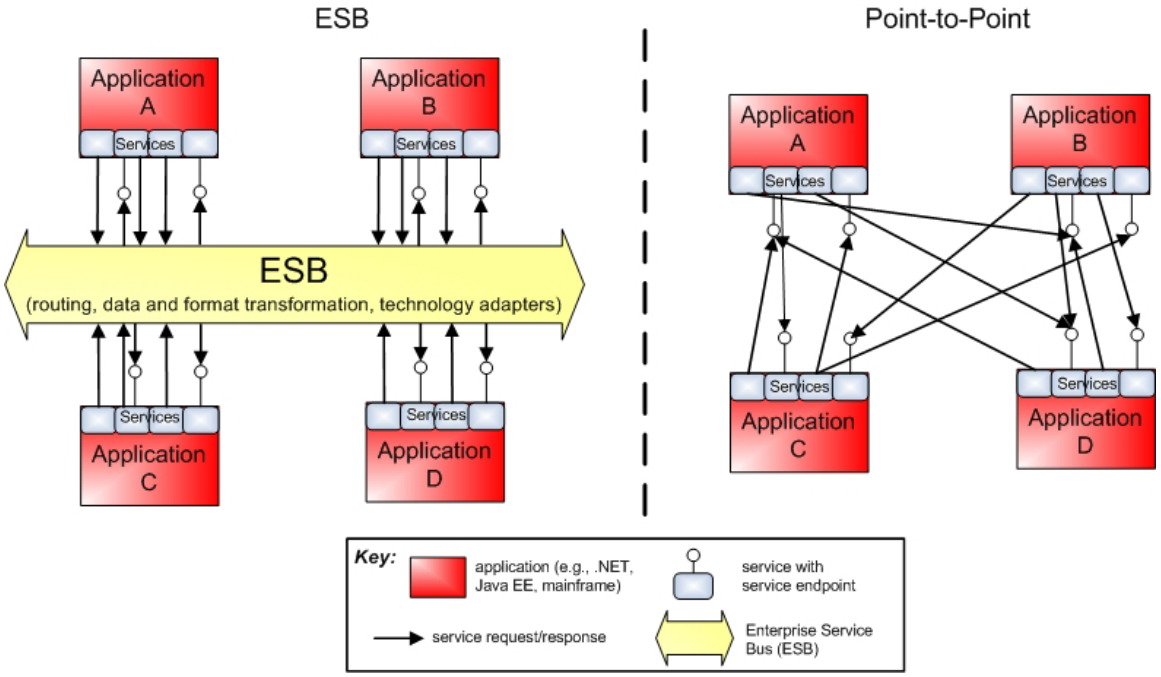

Figure 4: Simplified Comparison of ESB and Point-to-Point Integration Approaches

The term ESB is used interchangeably to refer to an architectural pattern and a product. While there is not an established industry standard that defines what constitutes an ESB, vendors and implementers have tried to identify some common capabilities that are outlined below:

- ESBs provide fundamental support for Web services .
- The ESB can route messages to one or more applications. Message routing that the ESB controls may be
  - fixed application-to-application
  - dynamic based on reading designated message content
  - dynamic based on system availability
  - dynamic based on load balancing
  - distribution from one source to many receivers (i.e., publish-subscribe)
  - consolidation of messages from multiple sources to one receiver (message aggregation)
- The ESB can transform data, including conversion of
  - data format (e.g., from a legacy application-specific, fixed-field record file format to a predefined XML schema)
  - business content (e.g., a part number in an enterprise resource planning (ERP) application to a different number in Web-based order-entry system)
  - multiplicity (i.e., splitting or combining separate messages)
- The ESB functionality can be distributed across multiple servers, which are centrally managed. Other hub-and-spoke solutions often mandate a single server.

- The ESB provides support for use of proprietary or custom adapters to connect to legacy and commercial off-the-shelf (COTS) applications.
- ESB products can support authentication, authorization, and encryption using multiple security standards such as WS-Security, Kerberos, and secure socket layer (SSL).
- ESB products typically provide advanced tooling (such as graphical document field mapping and routing definitions), integrated security, administration functions, and runtime monitoring services.

Primary architecture quality attributes that are addressed by an ESB include

- interoperability. An ESB allows connected applications with disparate technology and data formatting requirements to interoperate as service users and providers without invasive changes to each.
- modifiability. An ESB allows many (not all) types of changes or replacements of service providers without impacting the service users. For example, an ESB can be used to cross-reference IDs for products between applications or match date-and-time format standards without changing the source applications.
- extensibility. Compared to a point-to-point integration strategy, an ESB provides the ability to more easily add services as needed to meet changing business demands.

Adding an ESB to an SOA versus the use of direct point-to-point connections presents some architecture quality tradeoff considerations:

- Performance may be negatively impacted due to additional message hops and message transformation performed by the ESB.
- The overall system complexity and initial implementation cost are increased by adding an ESB to the architecture. Thus, adopting an ESB may not be feasible in environments with a small number of applications and services, or in projects with a tight schedule. An organization that adopts an ESB needs to
  - Define a long-term strategy comprising policies and standards for using the ESB, such as message format standards, connectivity and security standards, naming standards for service endpoints, queues, database connections, message schemas, and deployment files. These policies and standards are also important in direct point-to-point solutions, but become critical when there is a common backbone shared by all applications.
  - Establish processes to ensure that applications do not unjustifiably bypass the ESB.
  - Evaluate the ESB infrastructure and supporting platform to ensure that they provide mechanisms for transaction management, availability, logging and monitoring, error handling, scalability, and any other mechanism needed to meet the quality attribute requirements of the applications.
- Security administration mechanisms in an ESB environment can help to configure and manage access control of each connection to and from the ESB. On the other hand, content processed by the ESB may need to be selectively protected and exposed depending on routing and mapping requirements.

The choice among direct point-to-point, hub-and-spoke, or hybrid integration approaches is driven by factors such as

- current and planned number of integrated applications and technologies
- throughput and response time requirements of current and future integrated applications
- communication patterns (e.g., synchronous, message queues, publish-subscribe) and growing numbers of integrated services by current and future applications
- support requirements for new applications, business transactions, and data requirements
- adoption rate and maturity of new technologies and standards in the industry
- business, organizational, and regulatory dynamics (e.g., the speed with which acquired companies must be integrated)

4.3 BUSINESS PROCESS EXECUTION LANGUAGE (BPEL)

BPEL is a standard used to describe workflow-oriented business processes [OASIS 2006a]. A BPEL orchestration flow defines a business process through rules for coordinating the flow of data, interfaces to services (typically Web services) that the process exposes and uses, and provisions for handling exception conditions. Around the BPEL standard, vendors have created BPEL tools that enable nontechnical business programmers to devise workflows visually. Once interface descriptions for the participating services are in place, a BPEL tool can create BPEL code that describes the workflow. The BPEL language is XML-based and has primitives such as "receive," "reply," "throw," and "wait." The BPEL code is then posted to a BPEL engine (also called BPEL server) that runs on the application server. When the event that triggers the workflow happens, the BPEL engine coordinates the invocation of the services using the BPEL code as a script.

Capabilities typically provided within a BPEL orchestration implementation include the following types of processing:

- business process flow patterns of documents and service interactions. Operations that are part of a BPEL process flow may include
  - sequential flows of service invocations: calling services in a serial sequence
  - parallel service invocations: calling separate services in parallel, waiting for the responses before proceeding in a flow
  - request-reply correlation: issuing an asynchronous service call and correlating a separate service callback
  - timed wait: wait for a period of time for a service call response
- human-workflow-specific and business-process-specific interaction patterns. Examples include
  - work queue management (e.g., job prioritization, load balancing, automated reassignment)

- dual control (also known as double-check or four-eyes) approval workflow processes. In a procurement system for example, two levels of management could be required to approve the payment of invoices over a certain value.

- business process error handling. Example scenarios include
  - message delivery expiration
  - synchronous retry or abort upon failure
  - asynchronous retry compensating transaction upon failure
  - notification and heuristic resolution processes upon failure

Currently many SOA systems that implement business workflows are custom applications or are based on proprietary products. In the long term, it will be commonplace for medium to large SOA designs to rely on a BPEL engine for synchronizing internally and externally facing business processes and service connections.

Section 5.11 presents quality attribute considerations and design questions related to BPEL.

4.4 STATIC VERSUS DYNAMIC WEB SERVICES

To invoke a service provider, a service user needs to determine the interface of the service (operations available, expected input and output) and locate the actual service. For static binding, as shown in Figure 5, the service interface and location must be known when the service user is implemented or deployed. The service user typically has a generated stub to the service interface and retrieves the service location from a local configuration file. The service user can invoke the service provider directly, and no private or public registry is involved.

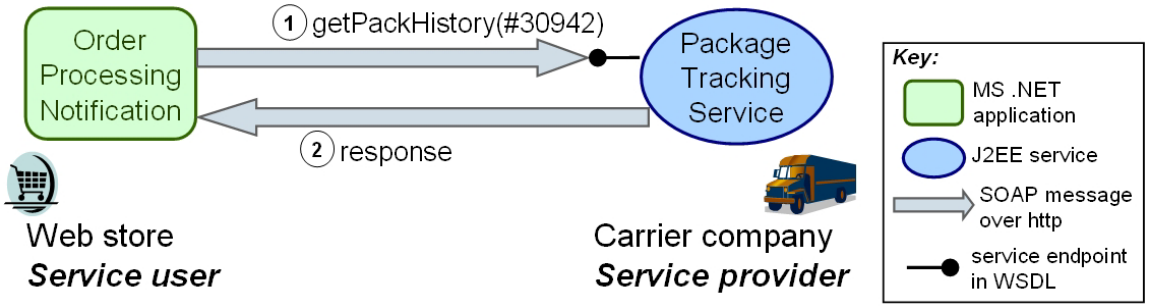

Figure 5: Static Binding Example

For dynamic Web services, as shown in Figure 6, a provider must register the service to a registry of services. The registry is queried by service users at runtime for the provider's address and the service contract. After acquiring the required information, the service user can invoke the operations of the service provider.

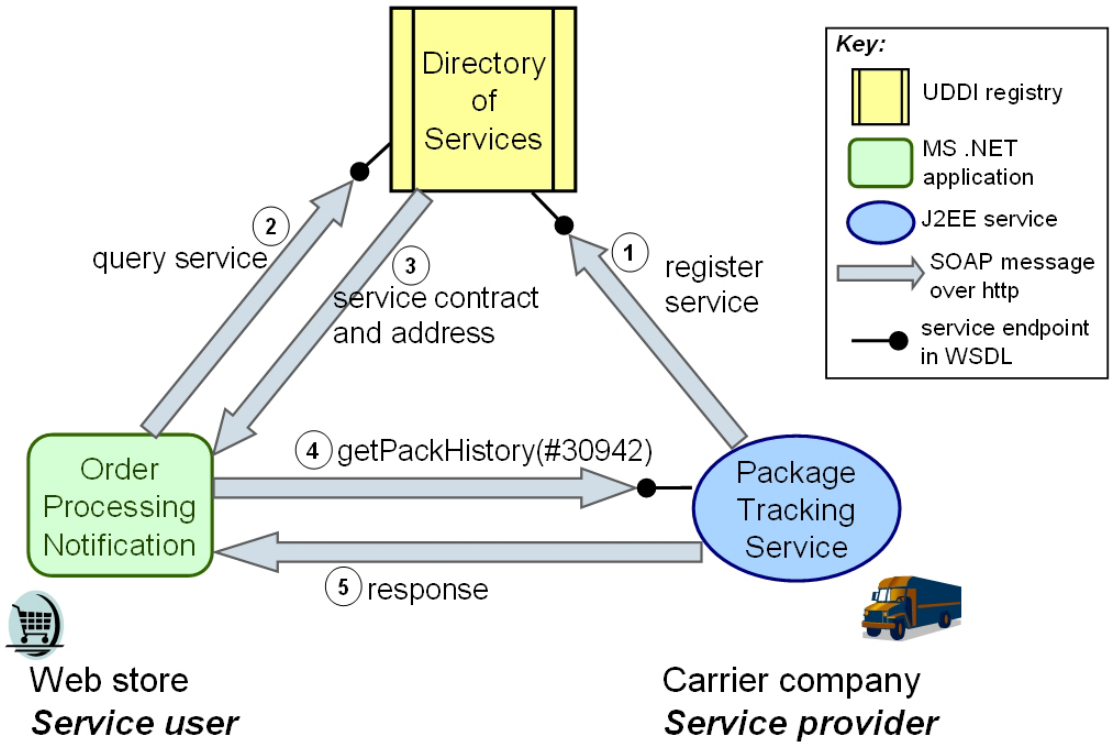

Figure 6: Dynamic Binding Example

Static binding results in a tighter coupling between service users and providers. Changes to the service location or contract can cause the service user to break at runtime when the service is invoked or during the marshalling and unmarshalling of the objects. The main advantage of static binding is better performance, because the communication to the registry is avoided. However, in some configurations where the registry is used to load-balance requests across a pool of service providers, the overall throughput can be better than the static binding alternative. Static binding is used frequently, because it is sufficient for most business scenarios and design solutions [Zimmermann 2003].

For dynamic binding the required information for invocation is obtained at runtime, thereby reducing the coupling between service users and providers. The service provider's location can change without affecting the service users. Multiple versions of interfaces can also be managed by the service registry and coexist in production. However, the flexibility given by dynamic binding requires service users and providers to have a predefined agreement on the syntax and semantics of the interfaces. Performance is negatively affected because of the interaction with the service registry. This performance overhead can be compensated by using the registry for load balancing. In fact the registry can have many purposes other than dynamic discovery of services. Section 5.9 discusses the capabilities, tradeoffs, and design questions involved in the use of a registry in an SOA solution.

4.5 EMERGING SOA-FOCUSED TECHNOLOGIES

A number of additional SOA standards, capabilities, best practices, products, and other technologies are emerging. Some architects latch on to the early use of new technologies and may misname, misapply, overuse, or abuse their concepts in project-risking ways. The evaluator may need to understand the impact of emerging technologies and raise tradeoff considerations during the SOA architecture evaluation. These considerations can be more critical depending on the organization's risk posture and technical capabilities. Below are some of the emerging areas to consider (full treatment of these trends in SOA is beyond the scope of this report):

- maturing and emerging WS-* standards, such as those related to transaction management, security, and reliable messaging (Some of these standards are discussed in Section 5.)
- the adoption of new language- and environment-specific standards, such as Service Component Architecture (SCA) [OSOA 2007], Service Data Objects (SDO) [SDO 2006], Windows Communication Foundation (WCF) [Microsoft 2007], and others
- Rich Internet Applications (RIAs) that directly use and combine service access from lightweight user clients
- use of EDA approaches to system design
- architectures and products that are based on Complex Event Processing (CEP) and Event Stream Processing (ESP)

5 SOA Design Questions That Affect Quality Attributes

Architectural design decisions determine the ability of the system to meet functional and quality attribute requirements. In the architecture evaluation, the architecture should be analyzed to reveal its strengths and weaknesses, while eliciting any risks. This section covers the following topics that are particularly relevant when designing SOA systems:

- target platform: Section 5.1
- synchronous versus asynchronous services : Section 5.2
- granularity of services: Section 5.3
- exception handling and fault recovery: Section 5.4
- security: Sections 5.5, 5.6, and 5.7
- XML optimization: Section 5.8
- use of a registry or services: Section 5.9
- legacy systems integration: Section 5.10
- BPEL and service orchestration: Section 5.11
- service versioning: Section 5.12

This list of design topics is not meant to be exhaustive or exclusive to SOA systems. It includes design concerns that the authors find to be more prominent in the SOA space but are sometimes overlooked in SOA projects. For each topic, we present potential evaluation questions that can be raised in an architecture evaluation and that will lead to a discussion of design alternatives and their implications. The recommendations are generic, and for each project, the implications of each alternative must be evaluated in light of all the existing factors. We relate the technical discussion to typical requirements that could be affected by the design decision. The typical requirements are further described as general quality attribute scenarios in Appendix A and referred to as P1, P2, …, A1, A2, and so forth.

5.1 WHAT IS KNOWN ABOUT THE TARGET PLATFORM?

Web services platforms can differ in their internal implementation and exposed functionality and qualities. The architect should be familiar with the target platforms, including the runtime environment for service user and service provider implementation components, the development environment, the network infrastructure, and the platforms used by external services.

5.1.1 Quality Attribute Discussion

In distributed application solutions, many quality concerns are primarily handled or strongly affected by the runtime environment. Examples include availability, throughput, interoperability, fault recovery, and data privacy. The ability to satisfy an interoperability scenario like I1 (see Appendix A) is in great part determined by compatibility issues between the two platforms involved.

Availability requirements as expressed in the general scenarios A3 and A4 are typically addressed by replication of software and hardware elements in the infrastructure. Replication mechanisms are usually provided by the platform and require some knowledge to be configured and tuned. Services provided by the infrastructure can partially handle security and reliability requirements similar to S1, S2, S5, R1, and R2.

5.1.2 Sample Evaluation Questions

These questions can be used in the architecture evaluation to probe the influence of the target platform in the achievement of the system requirements:

- Which Web services standards do the platforms of service users and providers implement? In particular, what WS-I profile do they implement? For example, IBM WebSphere V6.1 implements the WS-I Basic Profile V1.1. Let's say an application deployed to that platform needs to interact with an external service running on iPlanet Application Server V6.5, which is an old platform that is not compliant with any WS-I profile. Interoperability issues may arise due to compatibility problems between different versions of the WS standards implemented in these platforms.
- Are SOAP messages automatically translated to and from objects by the platform? If so, the service implementation does not need to implement that feature. However, the platform may not offer the most efficient translation, and that may affect performance when services handle large amounts of data.
- Which characteristics of the runtime platform may affect the security of the SOA solution? For example, if service users and service providers communicate via a virtual private network (VPN), the need for message-level security (see Section 5.5) may be relaxed.

Some questions are generic in the sense that they apply to more than SOA solutions:

- What properties in the runtime platform need to be tuned for the expected workload? The architect should help define how many instances of http listeners, database connections, server machines, and other architectural elements will be needed to meet the expected number of requests.
- What mechanisms are in place in the IT infrastructure to increase security, availability, reliability, and throughput? The hosting platform can totally or partially handle these properties using mechanisms such as replication of servers, firewalls, proxies, and load balancers.
- What support exists for monitoring and event data logging? These mechanisms allow taking measures, such as wait times, transaction volumes, and exception counts. These measures are important to oversee the system in production but are also useful at design time when technical experiments and prototypes are conducted for reliability and performance analysis.
- What are the known issues and technical limitations of the target platform version being used? Is the platform software maintained to patch levels to minimize vulnerabilities?
- Did the stakeholders who will create and deploy the system receive proper training on how to use the tools and frameworks needed to create and run the system? SOA development usually relies on several tools, such as ESB, object-to-WSDL translators, BPEL tools, and XML

schema generators. If developers are not familiar with the tools, they may waste time configuring the environment or manually implementing features that the tools automate.

5.2 SYNCHRONOUS OR ASYNCHRONOUS SERVICES?

Services may be provided through either synchronous or asynchronous interfaces in an SOA. Each option has pros and cons to consider, and the selection of a service interaction approach depends on a combination of business and application logic requirements, existing component capabilities (frequently, COTS or legacy applications support only one of the two service options, synchronous or asynchronous), and other architectural factors.

5.2.1 Quality Attribute Discussion

The choice between synchronous or asynchronous for each service user and service provider interaction can affect the system's ability to meet quality attribute requirements similar to the ones expressed by P1, P2, P3, R1, and I2 (see Appendix A). To aid in the evaluation process, the following table compares design tendencies in the use of synchronous versus asynchronous interactions as they relate to quality attributes.

Table 3: Comparison of Synchronous and Asynchronous Services

Quality Attribute	Synchronous Services	Asynchronous Services
Modifiability	☺ Typically simpler to develop and modify both service users and providers	☹ Implementation is frequently more complicated, because additional application logic is required to deal with the waiting and correlation of responses.
	☹ Behavior (e.g., timing and side effects) dependencies beyond the call interface make replacement more difficult. This tendency may result in brittle application designs.	☺ Lower coupling (applications and components can be more easily replaced with alternative modules)
	☹ It may be difficult to insert an ESB or other brokering software because of performance or other behavior dependencies.	☺ Ease of inserting ESB or other brokering software into conversations
Performance	☺ Expectation of and designed to achieve better responsiveness.	☹ Overhead of receiving asynchronous call responses and potential for delays in queue processing and failures
	☹ If used for a service request that could be processed asynchronously, the result is unnecessary blocking time.	☺ Allows background processing of service requests with no blocking time for service users

Table 3: Comparison of Synchronous and Asynchronous Services, Continued

Quality Attribute	Synchronous Services	Asynchronous Services
Scalability	☹ Typically lower scalability for large applications because of resource consumption and response time requirements for waiting service calls.	☺ Typically can achieve best scalability for SOA environments with large applications through time and server distribution of request processing
Reliability	☹ More susceptible to complex distributed failures because of direct interdependencies	☺ Better independent operation and fault-tolerance
	☺ Simpler error and exception handling designs	☹ Complex error/retry logic may be required.

5.2.2 Sample Evaluation Questions

When evaluating service interfaces for synchronous versus asynchronous design, the following questions help determine risks:

- Is the interaction between a given service user and provider synchronous or asynchronous? Not all service operations are suitable for asynchronous processing.
- How are architectural decisions about the use of synchronous versus asynchronous designs made? Are they driven by factors such as business requirements, legacy interface capabilities, and technology features? Not all operations are suitable for both synchronous and asynchronous processing. For example, processing an order in a Web store often can be handled asynchronously, but searching a catalog is usually a synchronous operation.
- Are services defined in a manner that will allow their use either synchronously or asynchronously? For example, are the interfaces stateless, and do they provide proper error-handling information?
- Are there intermediate hops in the flow of an asynchronous message? If so, how are the parties in the architecture identified and authenticated? Is data privacy enforced end to end?
- Does the asynchronous interface design correctly deal with error and retry logic?

5.3 COARSE- OR FINE-GRAINED SERVICES?

The granularity of a service refers to the scope of a service's functionality. A coarse-grained service typically consists of operations that require less communication and are designed to do more work with fewer service calls than fine-grained services. Service interface granularity has architectural and business implications, and is a critical factor for achieving certain quality attributes when implementing an SOA. Designing a service that is "right" grained depends primarily on how the service will be used, but the architect should also consider which quality attributes are most important to system stakeholders.

To illustrate the decision factors in selecting the granularity of a service, consider a simplistic example of a restaurant selling sandwiches. A sandwich seller offering a coarse-grained service pro-

vides sandwiches with several condiments included in the price. It does not make sense from a sellers' perspective of increasing efficiency to separately package and price every slice of bread, the meat, and the condiments as separate menu items. The efficiency at the cash register alone would be impacted; the cashier would be required to key in several items per sandwich. It also does not make sense for a sandwich seller to require consumers to purchase sandwiches in large quantities prepackaged on a pallet. Doing so would swing the coarse-grained/fine-grained pendulum too far for the taste of most consumers. While this analogy may seem obvious, aligning granular interfaces has been a common problem within distributed systems design. Poor alignment of interface utility versus functional requirements leads to failed or overly complicated and costly designs. Granularity choices are always somewhat subjective and require performance, security, and ease-of-use tradeoffs.

5.3.1 Quality Attribute Discussion

Coarse-grained services can improve application performance by reducing the number of messages required to complete a transaction. However, messages to and from coarse-grained services tend to be more verbose. Therefore, coarse-grained services have a negative impact in a performance scenario like P1 (see Appendix A), which is about the response time for a single request. However, they normally have a positive impact for scenarios like P2 (see Appendix A) that talk about the overall throughput of the system. A coarse-grained interface is less flexible from the perspective of the service user and can negatively impact general modifiability scenarios like M3 (see Appendix A). If interfaces are coarse-grained, localized interface changes that benefit a subset of the service users will impact more service users, and the overall cost of changes increases. Fine-grained services enable service reuse and composition by giving the clients more control over the steps of an operation.

Another quality that can be affected is security. A coarse-grained interface limits the potential entry points to a component, which may simplify the management of access rights. However, it does not allow for the flexible assignment of authorization for different operations.

Testability is also affected by the granularity of the service interface. In general, a coarse-grained interface is easier to test, because it limits the number of possible paths by consolidating the steps needed to process a user transaction under a single operation. Exposing more operations to the consumer causes a loss of control for the service provider. In a consumer Web site, the order service may require that a credit card is validated before an order can be submitted. A finer grained design exposing all steps as separate operations opens the door for the possibility of orders being submitted without the credit card's validation. The service user is now responsible for ensuring that steps are completed in the right order.

5.3.2 Sample Evaluation Questions

The following questions help determine whether there are any potential problems with the granularity of the services:

- What are the service network's bandwidth limitations? This can be potentially important to achieve the desired response time and throughput as described in general scenarios P1, P2,

and P3 (see Appendix A). If the interfaces are too fine grained, the transmission and processing of many small messages required to complete a task may be a performance risk. If the interfaces are too coarse grained, the overhead of parsing a large data set may be a performance risk.

- Do operations in the service interfaces map to transactional boundaries? Does each operation correspond to an atomic transaction, or can the transaction span the invocation of two or more operations? If services are stateless and each operation maps to an atomic transaction, it is easier to implement replication of the service provider components for improved availability and satisfy fault-recovery scenarios like R1 (see Appendix A).
- Are there ordering dependencies between operations? That is, is there a required order for the invocation of the operations? These questions impact the level of effort required to complete testing. A coarse-grained service interface that combines multiple ordered steps makes testing and implementation easier by reducing the number of possible test paths.
- How stable are the business processes represented by this architecture? Are certain services more likely to change than others? If services are more likely to change, it may make sense to have finer grained interfaces to promote the satisfaction of scenarios similar to M3 (see Appendix A). It is possible that changes that benefit a subset of consumers will impact all consumers. To explain this point more clearly, suppose that we have 6 operations exposed through a fine-grained interface. Each of these has 5 different users. If one operation is changed, 5 users will be affected. Then suppose that these 6 operations are now merged into 2 coarse-grained operations each with 15 different users. Now if one operation is changed, the number of affected users grows from 5 to 15.

5.4 WHAT ARE THE STRATEGIES FOR EXCEPTION HANDLING AND FAULT RECOVERY?

Achieving reliability, availability, and serviceability requirements is difficult in SOA systems. The system may involve heterogeneous platforms and protocols, as well as external services. A robust SOA-based architecture must deal with application and system failures at a variety of levels:

- system infrastructure (e.g., server and storage hardware; operating system and drivers)
- networking
- data services (e.g., relational database or LDAP directory server)
- middleware services (e.g., application server; queuing and messaging systems)
- service user and service provider implementation

The types of failures that can occur in SOA applications include

- the failure or resource exhaustion of an underlying component (e.g., out of memory or full queue)
- a formatting violation (e.g., invalid message against the XML schema)

- application business logic defects (e.g., a coding error that results in an unhandled null pointer exception)
- a failure of another application layer, or underlying data or legacy system service
- business rule failures, such as denial of access to a service based on user credentials and other factors, and violation of validation rules (e.g., insufficient funds, exceeded daily trading limit)

Error-handling strategies must address various failure-duration scenarios that include

- intermittent failures. In this case, the strategy may be to offer a "back-off and retry" option.
- semi-permanent and recoverable failures. The strategy may be to abort the transaction and notify users to retry the operation.
- permanent failures. The strategy may be to reroute transactions to an alternative service provider.

Establishment of proper debug, logging and tracing components, and standards help to detect failures and identify potential sources. Error-handling strategies should also manage the behavior of the system under failure modes. For example, in fail-safe mode, the failure should not be propagated to the point at which it compromises the whole system.

5.4.1 Quality Attribute Discussion

The quality and availability of diagnostic information to quickly isolate a root cause and the approach taken for error handling are key architectural areas of concern for distributed systems reliability, availability, and serviceability (RAS). If the system has requirements similar to the ones expressed in scenarios R1, R2, A1, A2, and S5, the architecture evaluation team should pay close attention to the fault-recovery and error-handling strategies. These strategies may have a negative impact on performance with overhead for persisting data for recovery, logging, and tracing.

5.4.2 Sample Evaluation Questions

The evaluation of exception-handling and fault-recovery strategies for an SOA should consider the following questions:

- Which types of failures is the system subject to?
- Do the distributed application components behave correctly together in the event of an anticipated failure? For example
  - Transaction rollback is performed to restore data to a consistent and correct state after a failure. This may require components to use distributed transaction protocols like XA or implement compensating transactions. Creating compensating transactions is challenging and more error-prone than relying on a transaction management service, but it may be the only alternative when third-party services are involved.
  - Failure notifications are generated to inform staff of the need for heuristic manual repair.
  - Under failure conditions, mechanisms in the architecture prevent additional damage to data and ensure correct behavior for concurrent users of the system (e.g., locking shared data).

  - Proper logging and audit-trail generation of failures is performed to allow rapid diagnoses and root cause repair for the issue. Designs should prescribe the tracing of key events, stack trace information, and other data relevant to the failed transaction.
- Does the design make proper use of facilities within the target platform for managing errors? For example
  - In Web services platforms, is the SOAP fault mechanism used?
  - In messaging systems, are abort/retry features and "dead-letter" handlers for asynchronous messages used?
  - In solutions that use an ESB or messaging system, are message format validation facilities in use? For example, the design must deal with "poison message" scenarios where a message causes the transaction to be aborted and is then returned to the queue for retry, resulting in an infinite loop.
  - In messaging systems, are persisted queues used for increased reliability?
- Do services provide idempotent and stateless operations where possible? A stateless and idempotent design is recommended to simplify error handling and recovery.
- Does the solution involve a messaging system, and are there cross-platform interoperability requirements? If so, does the platform support the WS-ReliableMessaging or WS-Reliability standards (see Section 4.1.3)?

5.5 HTTPS OR MESSAGE-LEVEL SECURITY?

A full range of architectural security concerns must be taken into consideration when evaluating an SOA environment, including the infrastructure (hardware, operating systems, networking), connected systems, authentication schemes, authorization, data privacy, non-repudiation, physical access, policy, and others. Full treatment of security in a distributed service-oriented environment is beyond the scope of this report. This section and the next two sections give special consideration to areas that have high impact: message-level data privacy, authentication, and authorization.

The https versus message-level security design aspect refers to the protection of messages exchanged between service users and service providers in an SOA solution using the Web services technology. The simplest alternative consists of using https (http over SSL) to secure the communication pipe at the transport level. In addition to encryption, https can optionally be used for authentication using digital certificates. Because there can be multiple hops between service user and provider, each point-to-point communication is secured separately, as illustrated in Figure 7.

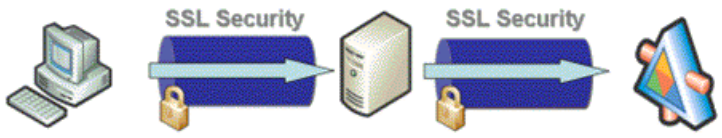

Figure 7: Https Security (from the work of Mitchell [Mitchell 2005])

Message-level security provides an end-to-end solution that protects the message itself, as illustrated in Figure 8. The actual content of the message is modified, and the security aspects are em-

bedded directly in the message. Standards such as WS-Security are needed to enable interoperable message-level security. At the message level, WS-Security describes how to authenticate services, how to ensure the integrity of services, and how to maintain confidentiality.

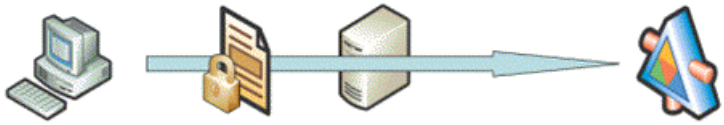

Figure 8: Message-Level Security (from the work of Mitchell [Mitchell 2005])

5.5.1 Quality Attribute Discussion

Embedding security as part of the message allows for a flexible end-to-end solution. For example, it allows the encryption of only portions of the message. Conversely, https encrypts the entire message and is only available from point to point at the transport layer. Thus, https does not protect the message at the application level and in locations where it is processed or stored, such as an ESB.

Message-level security is also extensible, because the platform configuration can be amended to include additional security credentials as needs change. The downside is that complexity is increased by requiring careful management of which parts of a message need to be secured. Interoperability is negatively impacted, because all parties that parse secure portions of the message need to support the security specifications in use. General interoperability scenarios like I1 are difficult to satisfy with the current state of the support for message-level security standards. On the other hand, https is simple to implement and highly interoperable. Also, performance is usually better when https is used instead of message-level security [Shirasuna 2004].

5.5.2 Sample Evaluation Questions

Deception and usurpation threats are common to distributed systems. Messages can be used to transmit viruses that usurp commands through shells or other mechanisms throughout the system. Common attacks include SQL injection, LDAP injection, and XQuery injection. These attacks can be used to change privileges, drop and alter tables, and change schema information [Lipson 2006]. Most of the following questions attempt to ascertain what mechanisms the architecture uses to deal with deception and usurpation threats:

- For each service user and provider interaction, does the architecture prescribe the use of https or message-level security?
- Does the architecture provide a mechanism (e.g., digital signatures) to ensure that a third party will not intercept and modify messages (tampering)? Which certificate authority is used for digital certificates? This question may affect general scenarios S2 and S5 (see Appendix A).
- Does your system interact with other systems that provide or require certificates? Are there known interoperability issues with respect to certificate exchange for the platforms involved? Note that not all certificate authorities are compatible. Scenarios like I1 may be affected (see Appendix A).

- What standards are you considering to support message-level security (e.g., WS-Security, XML Encryption, XML Signature)? The choice may affect interoperability scenarios like I1 (see Appendix A).
- How does the architecture protect against viruses, SQL injection attacks, and malicious scripts embedded in messages? The answer is related to general scenarios S2 and S5 (see Appendix A).
- How does the architecture handle message filtering? An example would be to block messages from certain IP addresses.
- Can the architecture support message-level security for REST and SOAP-based Web services ? This may affect interoperability as in general scenario I1 (see Appendix A).

5.6 HOW IS SERVICE AUTHENTICATION MANAGED?

There are SOA-specific authentication concerns that an evaluator should consider. These concerns are important even when the SOA solution does not use or provide services to external parties.

The authentication of participants in SOA integration scenarios may include requirements for authentication

- of an end user in a specific role
- of client applications
- across security realms or directory servers
- using mechanisms such as passwords, a Public Key Infrastructure (PKI), mutual authentication, tokens, or biometrics

5.6.1 Quality Attribute Discussion

When designing SOA systems, security tradeoff decisions are frequently required between business requirements and IT policy, not only for the service provider application but also for any connected service users. Authentication mechanisms are often required to satisfy security requirements similar to S3 and S4. However, adding levels of authentication to the architecture tends to negatively affect

- performance: overhead of authentication calls
- modifiability: additional code and deployment requirements to ensure security
- usability: complexity of managing and presenting credentials such as passwords, certificates, and tokens
- interoperability: incompatibility between authentication mechanisms supported by participant platforms

5.6.2 Sample Evaluation Questions

Some of the authentication-related questions to consider as an SOA architecture evaluator include

- What kind (e.g., LDAP based) and scope (e.g., enterprise wide, local) of security domain are going to be used for managing the identity for each participating system? How is the domain information shared across the participant applications that reside in different security domains?
- What authentication mechanisms are going to be used in each service user-provider interaction?
- What representation format is used to exchange security information between applications? The SAML [OASIS 2005] standard allows sharing security information about the participant's identity and access rights, and is used to implement a single sign-on (SSO) solution across services. Custom or proprietary approaches may limit interoperability with external services.
- If certificate or token-based services are used, do service users authenticate themselves to the service provider, does the service provider authenticate itself to service users, or is there mutual authentication?
- How is key management performed? Are there adequate policies and procedures for managing key exchanges and certificate signing? How will policies be enforced across participant systems?
- Does the architecture cover the full life cycle for end-user registration, validation, password reset, rights enablement, and other activities related to access control?
- Service implementations often need to access other resources, such as databases, other services, and components. How is the identity information used by service implementations, such as IDs and passwords, stored? Is it hard-coded (which is obviously bad) or centrally configured by an administrator?

5.7 HOW IS SERVICE ACCESS AUTHORIZATION PERFORMED?

As the adoption of an SOA approach grows within an organization and its external partners, the architect must comprehend the business process perspective and the technical security concerns to design a good authorization scheme and properly evaluate tradeoffs. It is challenging, because it requires an understanding of the access permissions required by different participants of the solution to different resources and operations available. Additionally, in certain industries there are regulatory-driven security concerns that must be accounted for when securing service interactions and data (e.g., HIPAA in healthcare or Sarbanes-Oxley for publicly traded U.S. firms.)

Additional challenges in implementing authorization and other security concerns in an SOA solution are limitations with respect to

- interoperability of security standards
- security implementations in legacy components that do not accommodate use as an external service (e.g., lack of support for an external LDAP directory server)
- identity management policy and technology across organizations

5.7.1 Quality Attribute Discussion

As outlined in Section 5.6.1 with respect to authentication, for authorization in SOA solutions, there can be security tradeoffs between business requirements and the IT policy for the server provider application and the service users. Also, adding levels of authorization to the architecture tends to negatively affect the same factors as authorization: performance, modifiability, usability, and interoperability.

5.7.2 Sample Evaluation Questions

In the evaluation of access authorization in an SOA environment, some of the areas of concern include

- What authorization mechanisms and standards (e.g., SAML) are going to be used to protect access to services? What kind of security domain will be used for managing permissions?
- Do various exposed service operations within the same service require different rights? Suppose that one is designing a service with three operations: browse catalog, place order, and update catalog. The first operation is open to any user, the second is open to registered users, and the third is open to administrators only. Depending on the authorization mechanism, different access rights within the same service are not easily implemented, and a better option is to implement multiple services.
- Is declarative authorization being used as opposed to programmatic authorization? Declarative security provided by the platform is preferable, because it separates security concerns from the business logic and may be changed at deployment time or runtime without modifying the source code. Programmatic security is also typically more error-prone. However, declarative security may not be viable where context information is needed to determine authorization rights (e.g., account information access may be restricted based on a combination of the time-of-day and prior management approval).

5.8 IS XML OPTIMIZATION BEING USED?

XML is the most common format for data representation in SOA solutions. It is flexible, extensible, widely adopted, and the underpinning for interoperability in most SOA technologies.

5.8.1 Quality Attribute Discussion

XML is text based and yields payloads that can be 10 to 20 times larger than the equivalent binary representation [Schmelzer 2002]. Three activities may be performed when processing XML documents, all of which are CPU and memory intensive: parsing, validation, and transformation. Strict performance requirements may call for XML optimization mechanisms to be discussed at the architecture level. Performance requirements, such as in scenarios P1, P2, and P3, are harder to meet when large amounts of data are transmitted and processed in XML format.

5.8.2 Sample Evaluation Questions

Questions to raise in the architecture evaluation include

- Is XML data compressed (e.g., Zip format)? The tradeoff between performance and interoperability is that compression requires that both use the same algorithm to interoperate.
- Does the hardware infrastructure include XML appliances? These network devices use specialized hardware and/or software to validate, transform, and parse XML messages faster. They have built-in support for standards that may include XML schema, XSLT, SOAP, and WS-Security.
- Can XML validation be turned off? That is possible when documents are known to be valid. If both service users and providers are developed by the same organization, versions of the XML documents that don't refer to a DTD or XML schema could be used in some cases.
- Are remote documents referenced in XML documents (e.g., an external schema) cached locally?
- Is the appropriate parsing model being used? When the XML document has to be accessed randomly or processed multiple times, DOM is more appropriate. When the elements in the documents have to be processed serially and only once, SAX yields better performance.
- Are validation and transformation of the XML data in a service request performed as soon as the request arrives? The early transformation allows smaller fragments of data in a format suitable for processing to be passed to the different modules that implement the service logic.

5.9 IS A SERVICE REGISTRY BEING USED?

In larger and rapidly changing SOA environments, it is difficult to manage the availability, capabilities, policies for use, and location of shared services. This difficulty results in the risk of quality failures. An SOA service registry provides the registration of services, management of metadata, and automation for the creation of and access to services. It is a central management service with the following capabilities:

- naming and location of service endpoints
- registration and querying of service descriptions including:
  - interface descriptions (WSDL) and XML schemas
  - metadata describing attributes of the service
  - security information about accessing the service
  - history and versioning information about the service
- dynamic service matching and binding
- support to the life cycle of services, including the following phases
  - inception: early business-function-level and (later) technical-level service definitions
  - design collaboration: coordination of interface design across applications
  - service provider implementation: defining the WSDL interface
  - service user implementation: retrieving WSDL and metadata for creating the client code
  - client provisioning: managing client access to services
  - testing and quality assurance

- deployment
- change management
- production
- versioning
- decommissioning

5.9.1 Quality Attribute Discussion

The implementation of a service registry primarily targets improvements to modifiability, manageability, and reliability of the overall SOA. The service user performance and maintainability may be negatively impacted by the overhead and complexity of the service lookup and related security. Interoperability issues may also exist with the use of an SOA registry because the standards are new.

5.9.2 Sample Evaluation Questions

The following questions help the architecture evaluator determine the registry's role and its effect on quality attributes of the overall architecture:

- Is a registry being used? If not, how do various parties using shared services know about the availability and capability of services? How is service information maintained to avoid unneeded duplication?
- What policies are in place to ensure the proper use of registries (versus circumvention using direct service location calls)?
- How is service metadata defined and managed within and outside of the registry? Are long-term considerations of future possible needs factored into the design?
- For which phases of the SOA application life cycle (inception through decommissioning) is the registry being used?
- How are service access control and change management policies governed? Are proper controls in place to balance security, modifiability, and compliance with IT and other standards? For example, new services from partners are only added to the registry after business, legal, security, and IT SLAs have been established. Updates to partner services then require versioning and adherence to the change management process.
- Is the registry being used for the dynamic routing of service calls (e.g., for failover, load balancing, and application partitioning)? If so, is the registry installation a single point of failure? Does it meet performance and failover time requirements?
- Is the registry interface based on standards like UDDI V3? Standards help development tools, administrative tools, and runtime components to interoperate with the registry.
- Does the registry provide validation or user notification for the addition or modification of services? These functions help keep the service aligned with standards and prevent the mismatch of client implementations.
- Is the registry public or private? Does the registry implementation properly handle the differentiation of internal and external services?

- Are there any technical requirements for the service users, such as "the service user must support https?"
- What types of service implementation policies are enforced by your organization?[3] Could these policies be enforced through the registry?
- Have you considered caching service locations to avoid calls to the registry and improve performance?

5.10 HOW ARE LEGACY SYSTEMS INTEGRATED?

There is typically more than one reasonable way to integrate a legacy system into an SOA environment. There are cost/benefit tradeoffs that an architect must weigh when selecting the integration strategy. Typical historical approaches to legacy system integration are

- direct database access
- batch-oriented file transfers
- database synchronization via extract, transform, and load (ETL) or custom tools
- direct API calls to legacy software interfaces
- messaging systems (e.g., IBM WebSphere MQ)
- screen scrapping
- Web services wrappers
- ESB with adapters for the legacy platform
- other application- and technology-specific gateways/bridges/adapters

5.10.1 Quality Attribute Discussion

A key goal of SOA is to improve the ability and agility to integrate new and existing systems as services. Most of the quality attribute discussion throughout this report related to SOA also applies to legacy systems, since they are simply other connected systems that provide services and run on different platforms. The integration of a legacy system is often expressed in interoperability scenarios similar to I2. The challenge is to conciliate diverging quality attribute requirements of the new and legacy systems.

5.10.2 Sample Evaluation Questions

The design considerations that drive the selection between alternative approaches for integrating a new system to a legacy system[4] include

- What mechanisms or strategies can be used to integrate the platforms of the new and legacy systems? How do these solutions compare in terms of

[3] For instance, a policy could be created to allow only SOAP bindings.

[4] For the purpose of the evaluation questions, a legacy system is one that does not directly support a Web Service interface.

  - complexity and cost of implementation vis-à-vis the available team schedule and skill set (e.g., Web services may not be readily supported or may be extremely cost-ineffective to establish in some legacy environments)?
  - performance, given the expected number of calls and desired response times?
  - security, given the access control and data privacy requirements in both systems?
  - reliability and support to distributed transactions?
- With respect to timeliness of executing operations or updating data sources, is there a mismatch between what is required in the new system and what is available in the legacy system (e.g., live real-time data sharing compared to daily batch updates compared to monthly shared updates)?
- What are the transactional access requirements for shared data in the legacy system (e.g., read-only versus concurrent read/write access by more than one application)? The number of calls to a legacy transaction may increase tremendously after the integration to the SOA architecture.
- What are the performance requirements for operations that involve interaction with the legacy system? For example, a synchronized full copy of order data from a legacy system of record may be needed for a consumer-facing Web application to provide fast access to order data.
- Are there SLAs between the new system and the legacy system covering properties such as communication performance, network security, availability, access control policies, and audit ability? For example, how do the availability requirements for the new system compare to the availability capabilities of the legacy system? Many legacy systems did not require 24-hour operation and provided batch windows during which transactional access was locked out.
- What is the anticipated lifetime of the legacy system? Is migrating the legacy system to the new platform an option? For example, it may be a better solution to migrate a legacy COBOL application to the new platform rather than to create a Web services wrapper around it in case the COBOL application is retired soon.
- Are there quality issues within legacy system source data? Frequently, loose data integrity constraints, manual data entry processes, and optimizations to save space resulted in poor data quality compared to expectations for modernized systems.
- Is the interface granularity of legacy components suitable for accessing them as services in an SOA?

5.11 IS BPEL USED FOR SERVICE ORCHESTRATION?

An overview of BPEL is provided in Section 4.3. This section outlines some of the factors that an evaluator should consider while reviewing the orchestration aspects of an SOA design.

5.11.1 Quality Attribute Discussion

An architecture evaluator for an SOA application should consider BPEL from multiple perspectives: as a modeling language, as an implementation language, and as the runtime engine. The primary architecture quality attributes affected by the use of BPEL are

- modifiability. Using BPEL to externalize process flow logic from source code allows easier implementation of business rules. Process workflow can be changed easily in the visual BPEL tool, which generates the BPEL code that will be deployed to the server.
- interoperability. The BPEL engine allows systems with disparate underlying platforms (e.g., Java and .NET) to interact through Web services technology. On the other hand, the BPEL standard is still emerging and inter-vendor interoperability limitations exist.
- performance. Additional software layers imposed by a BPEL engine, overhead to call the service interface, and the cost of BPEL code interpretation may negatively impact performance.
- cost. The overall system complexity, implementation cost, and total cost of ownership are increased. The increase may not be acceptable in simple environments where the cost of implementing and maintaining custom-coded process flow is less than the cost of implementing a BPEL-based application.
- reliability. Better defined and constrained sequencing of service interactions and exception handling results in more robust service integration behavior at an application level (when compared to custom-developed workflow applications). On the other hand, the additional complexity may cause reliability issues at a system/administrative level.

5.11.2 Sample Evaluation Questions

The questions below help evaluate design decisions related to BPEL:

- Does the BPEL engine make significant runtime status and issues available to support and maintenance staff?
- Are BPEL workflows focused on the business process and its requirements for modifiability? Some examples include
  - Are business rules and their parameters properly externalized for modification at runtime?
  - Does BPEL process design allow the easy replacement of service providers? This assumes that the service provider supplies the same, or an ESB-mapped, interface to the services used within the BPEL workflow definition.
  - Does the BPEL process and environment provide support for monitoring and logging event data to allow the measurement of business metrics, such as wait times, transaction volumes, and exception counts?
- Does each of the implemented BPEL processes properly deal with business and technical exception conditions and the need for compensating transactions?
- Does the BPEL engine generate audit trails with sufficient information to support transaction traceability and regulatory requirements? Does the audit trail information provide necessary detail to support non-repudiation requirements?
- Are BPEL processes designed with proper decoupling between services? For example, can one service in the process be changed without affecting every other service in the BPEL workflow?

- Is BPEL being circumvented for poor reasons or being overused for unintended purposes? Developers may try to circumvent the use of BPEL due to ignorance of its capabilities or for sociopolitical issues. In other cases, BPEL may be overused or misused because of a lack of understanding or the technologists' zeal to use something new.

5.12 WHAT IS THE APPROACH FOR SERVICE VERSIONING?

Services can be deployed and versioned independently of other system components. A new version of a service may be required not only when the interface syntax changes but also when any change in the service interface or implementation occurs that might impact consumer execution [Lublinsky 2007]. For example, if the new implementation changed the pre-condition for an existing operation, some service users' requests might be rejected. Another serious problem occurs when qualities of a service change, and the requests are processed. The resulting mismatch of assumptions between the provider and the user can lead to catastrophic failure. When the service is used by an unknown number of external service users, a common requirement is for old and new versions to coexist.

5.12.1 Quality Attribute Discussion

Service interfaces should be carefully designed to accommodate foreseeable service user requirements. But change is inevitable and a flexible and scalable versioning approach is required by modifiability scenarios like M2. The need to deploy and maintain multiple versions of different services increases the complexity of the configuration management and deployment processes. It may also cause a performance overhead with the introduction of intermediaries to route requests or resolve the address of the requested version.

5.12.2 Sample Evaluation Questions

The questions below help architects evaluate design decisions related to service versioning:

- What is the unit of versioning? Is it the whole service with all operations or individual operations within a service? Versioning operations requires deploying each operation with its own endpoint address. Service invocation becomes more complex, because the service user has to specify the service, the operation, and the version of the operation in the request. However, the impact of changes to service users is minimized, because only users of the altered operation are affected. Moreover, it allows different SLAs for different operations within the same service [Lublinsky 2007].
- What approach is used for schema versioning? Very often, service operations are defined with a standard signature equivalent to `XMLoutput serviceOperation(XMLinput)`. The input and output are defined by XML schemas. The signature of the operation never changes, but the XML schemas can change. A simple alternative for the schema versioning is to use the optional "version" attribute in the XML schema definition. Another approach is to create a new namespace for each new version. Other alternatives have different advantages and disadvantages [xFront 2007].

- How long should old versions of services/operations be preserved? Extending this period increases the effort to manage a large number of versions. Shortening the period imposes shorter deadlines for service users to perform upgrades [Lublinsky 2007].
- How are multiple versions concurrently deployed? One approach is to deploy all versions under the same endpoint address. Service requests indicate the required operation and version, and a routing component (e.g., an ESB) receives requests and dispatches them to the appropriate version of the service implementation. The benefit is that service addressing is simpler to implement in the service user. Another approach is to assign a different endpoint address to each version. The service user needs to resolve the endpoint address, typically by using a service registry, for the required version [Lublisnky 2007].

6 SOA Architecture Evaluation Example

The goal of this section is to show how the information discussed in Section 3 and the technical considerations presented in Sections 4 and 5 can be used to evaluate the architecture of an SOA system. We use a sample application and describe important aspects of performing an architecture evaluation. We follow the ATAM, briefly described in the first subsection. The ATAM analysis of the quality attribute scenarios gives insight into how well a particular SOA-based architecture satisfies the particular quality attribute goals of these scenarios and how certain quality attributes interact with each other in an SOA context. The results shown here are a subset of the information included in an ATAM report. We focus on individual scenarios and the architectural approaches used to build the sample system. When applicable, the scenario analysis provides references to the sections that address these architectural approaches.

6.1 ARCHITECTURE EVALUATION USING THE ATAM

What does it mean to say that a given software architecture is suitable for its intended purposes? At the SEI we believe that the suitability of an architecture is determined by the quality attribute requirements that are important to the stakeholders of a system. The ATAM relies on the elicitation of quality attribute scenarios from a diverse group of system stakeholders. The method was created to uncover the risks and tradeoffs reflected in architectural decisions relating to those quality attribute requirements. Quality attributes, also known as nonfunctional requirements, include usability, performance, scalability, reliability, security, and modifiability. Quality attribute scenarios give precise statements of usage, performance and growth requirements, various types of failures, and various potential threats and modifications [Bass 2003]. Once the important quality attributes are identified, the architectural decisions relevant to each high-priority scenario can be illuminated and analyzed with respect to their appropriateness [Barbacci 2003]. The resulting analysis yields

- risks: architectural decisions that might create future problems for some quality attribute. A sample risk: The current version of the Database Management System is no longer supported by the vendor; therefore, no patches for security vulnerabilities will be created.
- non-risks: architectural decisions that are appropriate in the context of the quality attribute that they affect. For example, the decision to introduce concurrency improves latency; the worst-case execution time for all threads is less than 50% of its deadline.
- tradeoffs: architectural decisions that have an effect on more than one quality attribute. For example, the decision to introduce concurrency improves latency but increases the cost of change for the affected modules.
- sensitivity points: a property of one or more components, and/or component relationships, critical for achieving a particular quality attribute requirement. For example, a de-

cision is made to choose REST over SOAP-based Web services for communication between service users and providers (see Section 4.1).

The ATAM method consists of the following nine steps:

1. **Present the ATAM:** The evaluation team presents a quick overview of the ATAM steps, the techniques used, and the outputs from the process.
2. **Present the business drivers:** The system manager briefly presents the business drivers and context for the architecture.
3. **Present the architecture:** The architect presents an overview of the architecture.
4. **Identify architectural approaches:** The evaluation team and the architect itemize the architectural approaches discovered in the previous step.
5. **Generate the quality attribute utility tree:** A small group of technically oriented stakeholders identifies, prioritizes, and refines the most important quality attribute goals in a utility tree format.
6. **Analyze the architectural approaches:** The evaluation team probes the architectural approaches in light of the quality attributes to identify risks, non-risks, and tradeoffs. To probe the architecture, they use quality attribute questions similar to the ones presented in Section 5.
7. **Brainstorm and prioritize scenarios:** A larger and more diverse group of stakeholders creates scenarios that represent their various interests. Then the group votes to prioritize the scenarios based on their relative importance.
8. **Analyze architectural approaches:** The evaluation team continues to identify risks, non-risks, and tradeoffs while noting the impact of each scenario on the architectural approaches.
9. **Present results:** The evaluation team recapitulates the ATAM steps, outputs, and recommendations.

These steps are typically carried out in two phases.[5] Phase 1 is architect-centric and concentrates on eliciting and analyzing architectural information. This phase includes a small group of technically oriented stakeholders concentrating on Steps 1 to 6. Phase 2 is stakeholder-centric, elicits points of view from a more diverse group of stakeholders, and verifies the results of the first phase. This phase involves a larger group of stakeholders, builds on the work of the first phase, and focuses on Steps 7 through 9 [Jones 2001].

A final report of the ATAM results include a summary of the business drivers, the architectural approaches, a utility tree, the analysis of each chosen scenario, and important conclusions. All these results are recorded visually, so stakeholders can verify the correct identification of the results.

[5] The ATAM also consists of a planning and preparation Phase 0. In this phase, the evaluation team looks at the existing architecture documentation to identify questions or areas of incompleteness. If the documentation is deemed insufficient to express a sound understanding of the multiple structures of the architecture, the evaluation does not proceed to Phase 1 (this constitutes a “no-go” decision).

6.2 SAMPLE APPLICATION

The system used as an example in this report is an adapted version of the Adventure Builder Reference application, which was developed in the context of the Java BluePrints program at Sun Microsystems [Sun 2007a]. This application was chosen because the functionality is easy to understand, and the source code, documentation, and other artifacts are publicly available for download. Also available is a book on Web services that explains the design and implementation of the application [Singh 2004]. We modified the architecture and made several assumptions about the business context and requirements of the system to make it a more interesting illustration of an SOA solution.

6.2.1 Functionality

Adventure Builder is a fictitious company that sells adventure packages for vacationers over the Internet. The system performs four basic operations (see Figure 9):

1. The user can visit the Adventure Builder Web site and browse the catalog of travel packages, which include flights to specific destinations, lodging options, and activities that can be purchased in advance. Activities include mountain biking, fishing, surfing classes, hot air balloon tours, and scuba diving. The user can select transportation, accommodation, and various activities to build his/her own adventure trip.
2. The user can place an order for a vacation package. To process this order, the system has to interact with several external entities. A bank will approve the customer payment, airline companies will provide the flights, lodging providers will book the hotel rooms, and businesses that provide vacation activities will schedule the activities selected by the customer.
3. After an order is placed, the user can return to check the status of his/her order. This is necessary because some interactions with external entities are processed in the background and may take hours or days to complete.
4. The internal system periodically interacts with its business partners (transportation, lodging, and activity providers) to update the catalog with the most recent offerings.

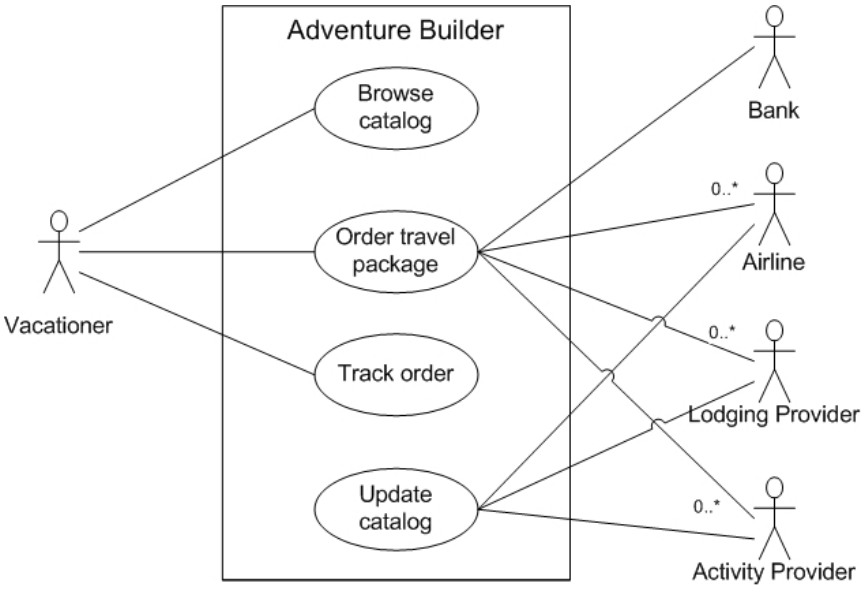

Figure 9: Basic Operations of Adventure Builder (UML Use Case Diagram)

6.2.2 Architecture Description

Figure 10 is a diagram of the top-level runtime view of the architecture. End users access the system using a Web browser. On the server side, the system is deployed as two distinct J2EE applications. One is called Consumer Web site and receives all requests from the users. Catalog browsing requests are processed by accessing the Adventure Catalog database. Purchase order and order tracking requests are processed by interacting with the other J2EE application, called Order Processing Center (OPC). OPC interacts with external service providers to fulfill order requests.

In the Web services technology, the entry point for the interaction between a service user and a service provider is called the Web services endpoint. In the diagram it is represented by a "lollypop" connected to the service provider. The endpoints are labeled with the name of the corresponding WSDL interface descriptions.

The external service implementation platform does not need to be known[6] to create the Adventure Builder architecture. That is why all external services are represented as gray rectangles in Figure 10. In reality, the various external service providers could use different technologies. Figure 11 depicts a possible scenario with exemplar external services.

[6] In practice there are platform-specific features that may hinder interoperability, as discussed in Section 5.

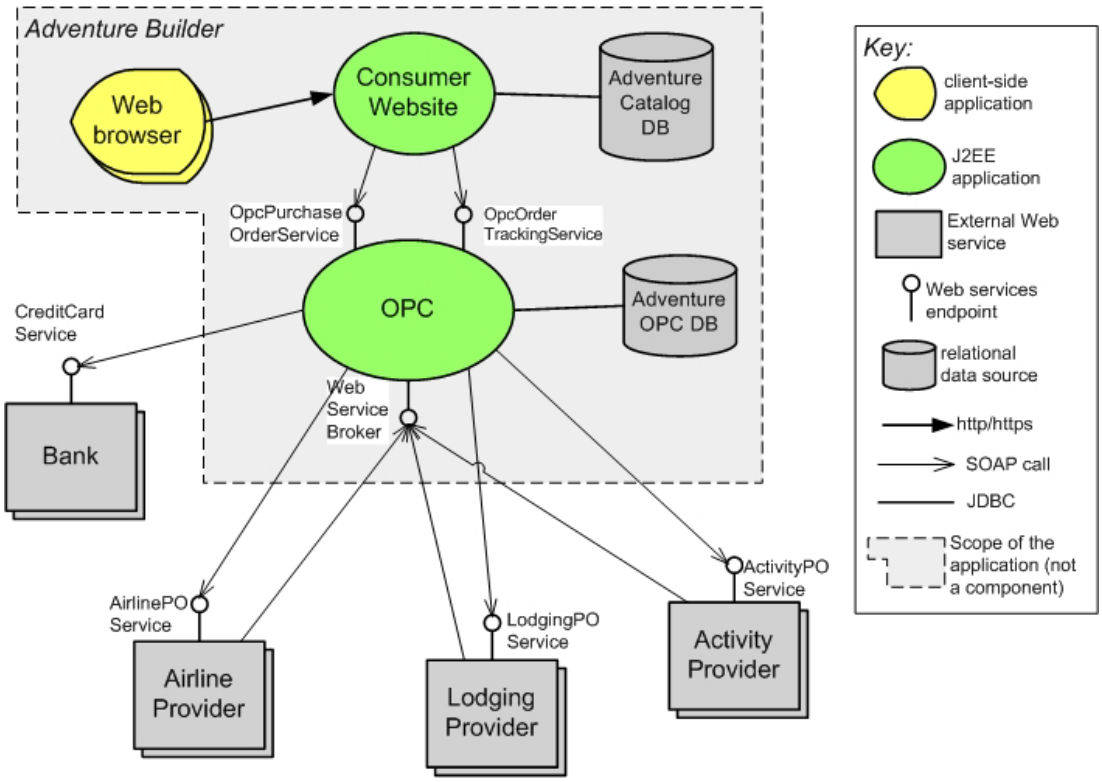

Figure 10: Top-Level Runtime View of the Adventure Builder Architecture

OPC acts as a service user when it interacts with Airline Provider, Lodging Provider, and Activity Provider. These interactions are asynchronous, because processing the requests can take a long time and the OPC application should not be blocked waiting for the results. For that reason, OPC also provides a callback endpoint (called Web Service Broker in the diagram). The airline, lodging, and activity external Web services interact asynchronously with OPC via the Web Service Broker endpoint to return the results of the original requests. The interaction sequence that takes place between service users and providers when a vacationer places an order is depicted in Figure 12.

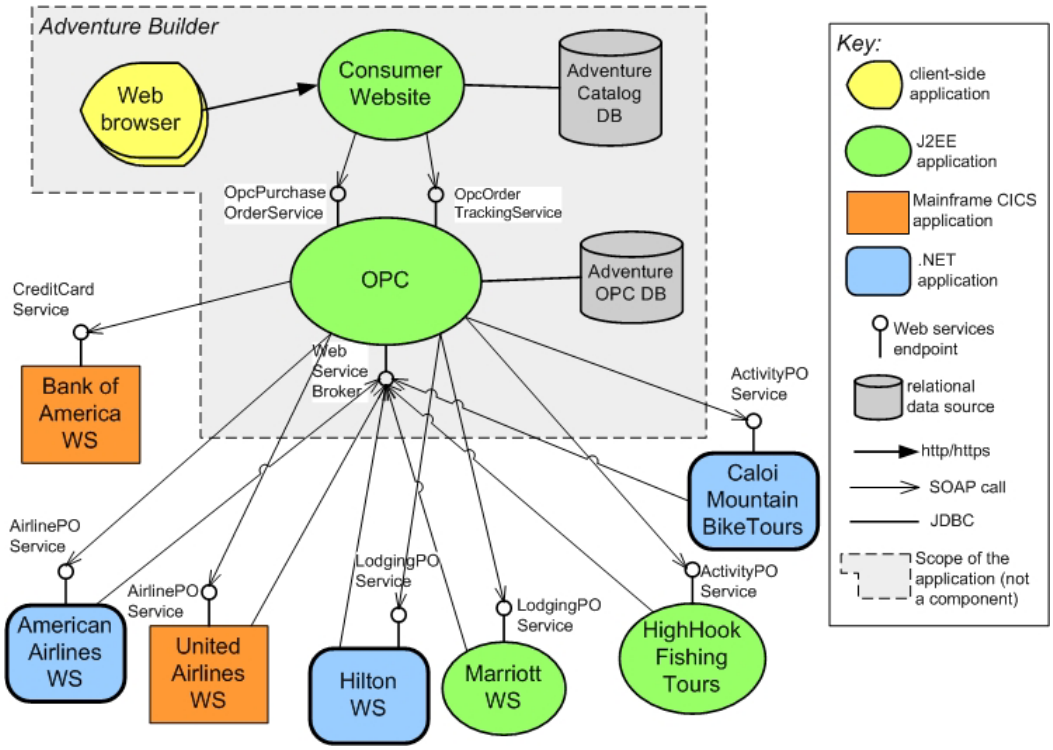

Figure 11: Runtime View with Exemplar External Services

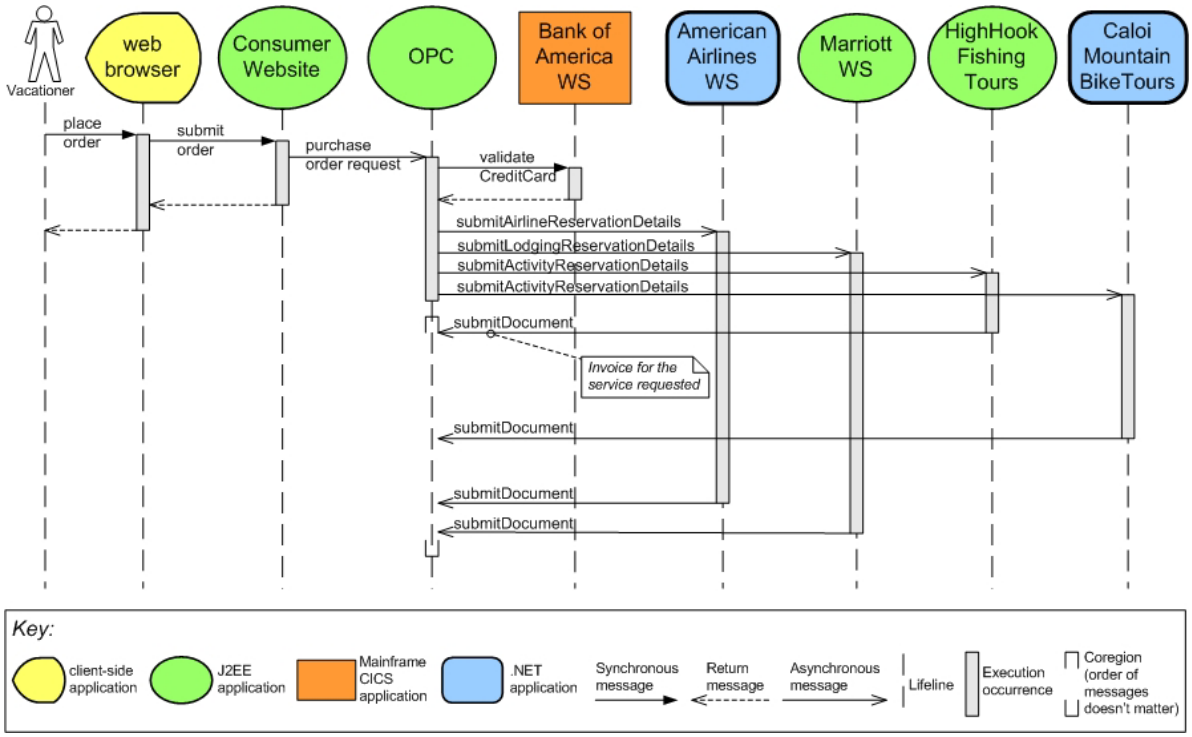

Figure 12: Sequence Diagram for Placing an Order

6.2.3 Quality Attribute Scenarios

Table 4 shows some quality attribute requirements specified using quality attribute scenarios for the Adventure Builder application. These scenarios are only a representative sample of possible quality attribute scenarios that may be relevant to an SOA-based architecture.

Table 4: Quality Attribute Scenarios for the Adventure Builder Application

Quality Attribute	Scenario
Scenario 1. Modifiability	• **(Source)** Business Analyst/Customer • **(Stimulus)** Add a new business partner (transportation, lodging, or activity provider) to use Adventure Builder's predefined Web services . • **(Artifact)** OPC • **(Environment)** Business partner familiar with the OPC interface and Web services technology • **(Response)** New business partner is added using Adventure Builder's Web services • **(Response Measure)** No more than one person-day of Adventure Builder team effort is required for the implementation (legal and financial agreements are not included).
Scenario 2. Modifiability	• **(Source)** Business Analyst/Customer • **(Stimulus)** Add a new airline provider that uses its own Web services interface. • **(Artifact)** OPC • **(Environment)** Developers have already studied the airline provider interface definition. • **(Response)** New airline provider is added that uses its own Web services . • **(Response Measure)** No more than 10 person-days of effort are required for the implementation (legal and financial agreements are not included).

Table 4: Quality Attribute Scenarios for the Adventure Builder Application, Continued

Quality Attribute		Scenario
Scenario 3.	Modifiability	• **(Source)** Business Analyst/Customer • **(Stimulus)** Add weather information for selected destinations that includes average daily temperature and average monthly precipitation. • **(Artifact)** Consumer Web site • **(Environment)** Developers familiar with the interface definition of the weather service • **(Response)** The external service that provides weather information is integrated with the system, and the new feature is available to Adventure Builder users. • **(Response Measure)** No more than two person-months of effort are required for the implementation.
Scenario 4.	Performance	• **(Source)** User • **(Stimulus)** User submits an order for a package to the Consumer Web site. • **(Artifact)** Adventure Builder system and the Bank • **(Environment)** Normal operation • **(Response)** The Consumer Web site notifies the user that the order has been successfully submitted and is being processed by the OPC. • **(Response Measure)** The system responds to the user in less than five seconds.
Scenario 5.	Reliability	• **(Source)** External to system • **(Stimulus)** The Consumer Web site sent a purchase order request to the OPC. The OPC processed that request but didn't reply to Consumer Website within five seconds, so the Consumer Web site resends the request to the OPC. • **(Artifact)** Adventure Builder system • **(Environment)** Normal operation • **(Response)** The OPC receives the duplicate request, but the consumer is not double-charged, data remains in a consistent state, and the Consumer Web site is notified that the original request was successful. • **(Response Measure)** In 100% of the transactions

Table 4: Quality Attribute Scenarios for the Adventure Builder Application, Continued

Quality Attribute		Scenario
Scenario 6.	Reliability	• **(Source)** System failure in the OPC • **(Stimulus)** The OPC sends a request to the bank to charge a credit card for a purchased travel package; before receiving the reply from the bank, the OPC crashes. • **(Artifact)** OPC and bank service • **(Environment)** Failure mode • **(Response)** The system recovers in a correct and consistent way, and the credit card is charged only once. • **(Response Measure)** In 100% of the cases
Scenario 7.	Availability	• **(Source)** Internal to the system • **(Stimulus)** Fault occurs in the OPC • **(Artifact)** OPC • **(Environment)** Under normal operation • **(Response)** The system administrator is notified of the fault; the system continues taking order requests; another OPC instance is created; and data remains in consistent state. • **(Response Measure)** The fault is detected, and failover action is taken within 30 seconds.
Scenario 8.	Security/ Availability	• **(Source)** External to system • **(Stimulus)** The OPC experiences a flood of calls through the Web Service Broker endpoint that do not correspond to any current orders. • **(Artifact)** OPC • **(Environment)** Normal operation • **(Response)** The system detects the abnormal level of activity and notifies system administrators. • **(Response Measure)** The system continues to service requests in degraded mode.
Scenario 9.	Security	• **(Source)** User • **(Stimulus)** Credit approval and payment processing functions are requested for a pending order. • **(Artifact)** OPC and the Bank's service • **(Environment)** Normal operation • **(Response)** The credit approval is completed securely and cannot be refuted by either party. • **(Response Measure)** In 100% of the transactions

6.3 ARCHITECTURAL APPROACHES

In an ATAM evaluation, the architectural approaches are identified during Steps 3 (Present Architecture) and 4 (Identify Architectural Approaches). Hub-and-spoke is the overarching architectural approach of the Adventure Builder application. Although an ESB product is not used, the OPC has a centralized workflow manager that contains all process rules and flow logic to coordinate the processing of customer orders. The individual "spokes" in the OPC (external business partners) execute their part of the business functionality and have no need to know the details of the overall process. The use of the "hub" as an active mediator reduces the dependencies between the "spokes" to promote modifiability. Most changes to any single "spoke" should be localized and should only require changes to the "hub."

The OPC uses SOAP-based Web services to communicate with the Consumer Web site and external business partners. Web services promote interoperability with a wide array of technologies deployed by potential partners. The Web services interface design also promotes decoupling between the OPC and the software of the business partners.

Web services for the communication between the Consumer Web site and the OPC enables the Web site to be hosted outside the firewall in the demilitarized zone, while the OPC module remains inside the firewall. The communication is handled through an http port available on the firewall. The choice of Web services allows the Consumer Web site and the OPC to be deployed on different hardware and software platforms.

As mentioned previously, an evaluation focused on service integration does not cover every important architectural decision. For example, the architectural pattern used in the Consumer Web site is the Model-View-Controller (MVC) pattern to promote modifiability. This design decision should also be analyzed to identify risks and tradeoffs.

6.4 ARCHITECTURAL ANALYSIS

The analysis prescribed in the ATAM is not meant to be precise and detailed; it does not provide numerical values for different qualities. The key is to elicit enough architectural information to identify risks, which result from the correlation between the architectural decisions and their effect on quality attributes. The evaluation team typically probes the architectural approaches used to address the important quality attribute requirements specified in the scenarios. The goal is to assess whether these quality attribute requirements can be met. The evaluation is done to capture the architectural approaches and identify their risks, non-risks, sensitivities, and tradeoffs [Clements 2002b]. The analysis of some of the scenarios from Section 6.2.2 follows:

Table 5: Architectural Analysis for Scenario 2

Analysis for Scenario 2	
Scenario Summary	A new airline provider that uses its own Web services interface is added to the system in no more than 10 person-days of effort for the implementation.
Business Goal(s)	Permit easy integration with new business partners.
Quality Attribute	Modifiability, interoperability
Architectural Approaches and Reasoning	• Asynchronous SOAP-based Web services (see Sections 4.1 and 5.2) • Interoperability is improved by the use of document-literal SOAP messages (see Section 4.1) for the communication between OPC and external services. • Adventure Builder runs on Sun Java System Application Server Platform Edition V8.1. This platform implements the WS-I Basic Profile V1.1, so interoperability issues across platforms are less likely to happen (see Section 5.1).
Risks	• The design does not meet the requirement in this scenario, because it assumes that all external transportation providers implement the same Web services interface called AirlinePOService (as shown in Figure 10 and Figure 11). The design does not support transportation providers that offer their own service interface.[7]
Tradeoffs	• The homogenous treatment of all transportation providers in OPC increases modifiability. However, intermediaries are needed to interact with external providers that offer heterogeneous service interfaces, as in this scenario. These intermediaries represent a performance overhead, because they may require routing messages and extensive XML processing.

[7] An ESB (see Section 4.1) could be a solution to the integration with this new airline provider and any other external services in the future.

Table 6: Architectural Analysis for Scenario 4

Analysis for Scenario 4	
Scenario Summary	User submits an order for a package to the Consumer Web site. The system responds to the user in less than five seconds.
Business Goal(s)	Provide satisfactory user experience.
Quality Attribute	Performance
Architectural Approaches and Reasoning	• The use of document-literal SOAP results in better performance, because there is no overhead associated with encoding (see Section 4.1). • Static Web service (see Section 4.4) prevents the overhead of looking up a registry. • The Web services were designed around the documents handled, such as purchase orders and invoices. The OPC Purchase Order Service interface (see Figure 10) is coarse grained in the interest of improving system performance (see Section 5.3). This interface reduces the overhead of calling a fine-grained service for each step of a business process. • The OPC interacts with the Bank in a synchronous fashion (see Section 5.2). The charge is authorized quickly so that processing of the order may proceed. Then the OPC sends requests to transportation, lodging, and activity providers, which will later respond through the Web Service Broker callback endpoint. These requests are sent asynchronously to improve scalability and throughput and also because of the nature of the legacy systems supporting this interface.
Risks	• The Adventure Builder architects have no control over the latency of external service providers.[8]
Tradeoffs	• Using Web services for communication with the Consumer Web site and external entities promotes loose coupling and interoperability. However, the overall latency of requests increases because of the overhead required for translating among WSDL, Java, and the other XML processing involved (see Section 5.8).

[8] An SLA should be negotiated to mitigate this risk to some extent.

Table 7: Architectural Analysis for Scenario 5

Analysis for Scenario 5	
Scenario Summary	The Consumer Web site sent a purchase order request to the OPC. The OPC processed the request but didn't reply to the Consumer Web site within five seconds. So the Consumer Web site resends the request to the OPC. The OPC receives the duplicate request, but the consumer is not double-charged; data remains in a consistent state; and the Consumer Web site is notified that the original request was successful.
Business Goal(s)	Provide satisfactory user experience by preventing overcharges and double booking.
Quality Attribute	Reliability
Architectural Approaches and Reasoning	• No transactions are distributed across multiple databases. Each piece of an order is a separate transaction. The centralized workflow manager in the OPC contains the state of each order during processing, and the database supports atomic transactions. • A correlation identifier to match requests and asynchronous responses allows idempotent endpoints for service providers that update or change state (see Section 5.4). The use of idempotent endpoints promotes reliability.
Risks	None
Tradeoffs	• A single database promotes reliability and reduces complexity at the expense of availability by introducing a single point of failure in the OPC. • The use of idempotent endpoints promotes reliability but imposes performance overhead and adds complexity to implementation.

Table 8: Architectural Analysis for Scenario 9

Analysis for Scenario 9	
Scenario Summary	Credit approval and payment processing functions must be secure and provide for non-repudiation.
Business Goal(s)	• Provide customers, Adventure Builder's business, and business partners with confidence in security. • Meet contractual, legal, and regulatory obligations for secure credit transactions.
Quality Attribute	Security
Architectural Approaches and Reasoning	• Adventure Builder uses SSL mutual authentication (see Section 5.6). OPC and the Bank exchange digital certificates through an SSL handshake. Communication is encrypted using https. • Declarative authorization is used to set authorization rights (see Section 5.7).
Risks	• If certificate management is not done carefully, modifiability and interoperability will be negatively impacted. • The Adventure Builder system has only contractual (not technical) enforcement of information security management stored in partner systems.
Tradeoffs	• Implementing SSL mutual authentication adds complexity, hence increasing the time and costs of development and maintenance. • Encryption of messages exchanged with external partners adds some performance overhead.

7 Conclusion

SOA is a widely used architectural approach for constructing large distributed systems, which may integrate several systems that offer services and span multiple organizations. In this context, it is important that technical aspects be considered carefully at architectural design time. SOA systems are often part of technologically diverse environments, which involve numerous design considerations. Many of those considerations were covered in this report. In a software architecture evaluation, we weigh the relevance of each design concern only after we understand the importance of each quality attribute requirement.

Because decisions about SOA tend to be pervasive and have a significant and broad impact on business, performing an early architecture evaluation is particularly valuable and recommended. The ATAM method can be used as-is to evaluate an SOA system. Architecture evaluators should pay special attention to solutions that use dynamic binding to allow alternative execution pathways and different ordering of requests—quality attributes are harder to predict and analyze in these solutions. The stakeholder categories do not seem to be particularly different for SOA systems, but the list is more dynamic, especially when external services are part of the solution. In the architecture description, the runtime view best shows the SOA approach.

Because SOA involves the connectivity of multiple systems, business entities, and technologies, its overall complexity and the political forces involved need to be factored into architecture trade-off considerations more than in single-application designs where technical concerns predominate. Balancing SOA aspects against other software architecture concerns is particularly challenging in an SOA software architecture evaluation. Frequently, only part of a system is SOA-based, so the evaluation needs to address both SOA-specific issues and those relevant to other combined approaches.

The technology is changing, so the technical discussion in this report will require constant updates. Nonetheless, some of the issues discussed will remain valid and indeed were valid 20 years ago when distributed systems became a reality.

8 Feedback

The authors will continue to investigate ways to improve software architecture activities in the context of SOA systems and are interested in feedback from the community. SOA represents a fast-changing technology space, and this report will require occasional updates as standards and best practices mature. If this report contains information that you deem inaccurate or outdated, if you want to suggest additional topics, or if the report was useful to you, please let us know:

Paulo Merson – pfm@sei.cmu.edu

Phil Bianco – pbianco@sei.cmu.edu

Rick Kotermanski – rek@summa-tech.com

Appendix A Sample SOA General Quality Attribute Scenarios

In order to evaluate the quality of any system, we first need to characterize the various quality attribute requirements applicable to the system. Quality attribute scenarios serve this purpose. For the same reason that use cases can describe functional requirements, quality attribute scenarios are used to specify quality attribute requirements. We enumerate a collection of quality attribute *general scenarios* for seven quality attributes that are important to consider when using a service-oriented approach: modifiability, performance, availability, security, reliability, interoperability, and testability. These general scenarios are used through sections 4 and 5 to illustrate how qualities are affected by architectural decisions. These scenarios are "general" in the sense that they are system independent [Bass 2001].

Performance

P1 - A sporadic request for service ‘X’ is received by the server during normal operation. The system processes the request in less than ‘Y’ seconds.
P2 - The service provider can process up to ‘X’ simultaneous requests during normal operation, keeping the response time on the server less than ‘Y’ seconds.
P3 - The roundtrip time for a request from a service user in the local network to service ‘X’ during normal operation is less than ‘Y’ seconds.

Availability

A1 - An improperly formatted message is received by a system during normal operation. The system records the message and continues to operate normally without any downtime.
A2 - An unusually high number of suspect service requests are detected (denial-of-service attack), and the system is overloaded. The system logs the suspect requests, notifies the system administrators, and continues to operate normally.
A3 - Unscheduled server maintenance is required on server ‘X.’ The system remains operational in degraded mode for the duration of the maintenance.
A4 - A service request is processed according to its specification for at least 99.99% of all requests.
A5 - A new service is deployed without impacting the operations of the system.
A6 - A third-party service provider is unavailable; modules that use that service respond appropriately regarding the unavailability of the external service; and the system continues to operate without failures.

Security

S1 - A third-party service with malicious code is used by the system. The third-party service is unable to access data or interfere with the operation of the system. The system notifies the system administrators.
S2 - An attack is launched attempting to access confidential customer data. The attacker is not able to break the encryption used in all the hops of the communication and where the data is persisted. The system logs the event and notifies the system administrators.
S3 - A request needs to be sent to a third-party service provider, but the provider’s identity can not be validated. The system does not make the service request and logs all relevant information. The third party is notified along with the system administrator.

S4 - An unauthorized service user attempts to invoke a protected service provided by the system. The system rejects the attempt and notifies the system administrator.
S5 - An attacker is modifying incoming service requests in order to launch an attack on the system infrastructure. The system identifies and discards all tampered messages, logs the event, and notifies the system administrators.
S6 - An attacker attempts to exploit the service registry in order to redirect service requests. The service registry denies access to information in the registry, logs the event, and notifies the system administrators.

Testability

T1 - An integration tester performs integration tests on a new version of a service that provides an interface for observing output. 90% path coverage is achieved within one person-week.

Interoperability

I1 - A new business partner that uses platform 'X' is able to implement a service user module that works with our available services in platform 'Y' in two person-days.
I2 - A transaction of a legacy system running on platform 'X' is made available as a Web service to an enterprise application that is being developed for platform 'Y' using the Web services technology. The wrapping of the legacy operation as a service with proper security verification, transaction management, and exception handling is done in 10 person-days.

Modifiability

M1 - A service provider changes the service implementation, but the syntax and the semantics of the interface do not change. This change does not affect the service users.
M2 - A service provider changes the interface syntax of a service that is publicly available. The old version of the service is maintained for 12 months, and existing service users are not affected within that period.
M3 - A service user is looking for a service. A suitable service is found that contains no more than 'X' percentage of unneeded operations, so the probability of the service provider changing is reduced.

Reliability

R1 - A sudden failure occurs in the runtime environment of a service provider. After recovery, all transactions are completed or rolled back as appropriate, so the system maintains uncorrupted, persistent data.
R2 - A service becomes unavailable during normal operation. The system detects and restores the service within two minutes.

Appendix B Glossary of Technical Terms and Acronyms

ATAM	The Architecture Tradeoff Analysis Method (ATAM) [Kazman 2000, Clements 2002b] is an architecture evaluation method developed at the SEI that consists of nine steps and identifies risks related to the ability of an architecture to meet its quality attribute requirements. For more details, see Section 6.1.
BPEL	Business Process Execution Language (BPEL) is a standard [OASIS 2006a] used to describe workflow-oriented business processes. For more details, see Section 4.3.
Compensating transaction	A compensating transaction undoes the effect of a previously committed transaction.
Conversational state	Refers to data that a server-side component stores on behalf of the client component across two or more calls. For example, in a Web store solution, the conversational state of a server-side component can include the contents of the shopping cart. This state should be preserved for each user across multiple requests.
CORBA	The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in different programming languages and running on different platforms to interact.
Data marshalling	See Serialization.
Dead-letter	In a messaging system, messages that are not delivered are recorded in a dead-letter queue. A few reasons for failed delivery include network failures, a queue being full, and authentication failure.
Document-literal	See the description on page 14.
DOM	Document Object Model (DOM) is a W3C standard for representing and manipulating XML data as a set of objects organized in a tree data structure.
DTD	The Document Type Definition (DTD) is a type of document that contains a set of declarations to define the structure of XML files. It is used for XML validation.
EAI	Enterprise Application Integration consists of software and architectural principles that allow for the integration of applications. EAI attempts to provide real-time access to data and processes with minimal changes to the existing applications and their underlying data structures.
ESB	Enterprise Service Bus is an architectural style that creates a "universal integration backbone" that provides infrastructure services to other services or applications to promote a consistent approach to integration while reducing the complexity of the applications or services being integrated. See Section 4.2.

ETL	ETL (extract, transform, and load) is the data warehousing process for extracting outside data while transforming it to conform to organizational needs and then loading it into the data warehouse.
Jini	Jini is a technology proposed by Sun Microsystems to create distributed systems that consist of cooperating services.
LDAP	Lightweight Directory Access Protocol (LDAP) is an application protocol for querying and modifying directory services running over TCP/IP.
Marshalling	See Serialization.
Object-to-WSDL translation	Process where a development tool takes an object interface definition (e.g., a Java class or interface) and generates the corresponding WSDL definition.
REST	See the description on page 16.
RPC-encoded SOAP	See the description on page 13.
SaaS	Software as a service (SaaS) is a software delivery model where customers don't own a copy of the application. Instead of paying for a software license, customers pay for using the software, which is available via the Web.
SAML	Security Assertion Markup Language (SAML) is an XML standard for exchanging authentication and authorization data between security domains.
SAX	Simple API for XML (SAX) is a common mechanism to parse XML documents serially. Events are generated for each element parsed in the XML document.
Screen scraping	Technique where a program (screen scraper) extracts data from the display output of another program and sometimes also sends data to that program emulating keyboard data entry.
Serialization	Serialization refers to the process of transforming the memory representation of an object to a data format suitable for storage or transmission. Serialization is also called marshalling, and the opposite operation is called deserialization or unmarshalling.
SOAP	Simple Object Access Protocol (SOAP) is the XML-based protocol used for the message exchange between service users and providers when Web services technology is used. SOAP is a W3C standard and the current version is 1.2 [W3C 2003].
SOAP fault	The body of the SOAP messages may contain a standard element called *fault* that carries information about an error that occurred.
UDDI	Universal Description Discovery and Integration is a platform-independent, XML-based registry that allows service providers to list their services and define how service consumers can locate and interact with those services. UDDI is at the core of Web services standards and is sponsored by OASIS.

Unmarshalling See Serialization.

WSDL Web services Description Language (WSDL) is the XML-based language used to specify service interfaces in the Web services technology [W3C 2007]. Services are described as a set of endpoints.

WS-I WS-I is an open industry organization that promotes Web services interoperability. For more information, go to http://www.ws-i.org.

WS-I profile A profile specifies a list of Web services standards at specific versions. It also contains normative statements that impose ("must") or recommend ("should") restrictions on how each standard can be used. Vendors of Web services products that implement the standards and normative statements can claim conformance to a profile.

As an example, the Basic Profile V1.1 requires SOAP V1.1, WSDL V1.1, and UDDI V2. One of the normative statements is: "R1107 A RECEIVER MUST interpret a SOAP message as a Fault when the `soap:Body` of the message has a single `soap:Fault` child."

WS-Security WS-Security is an OASIS standard for applying security to Web services [OASIS 2004b]. It defines a set of SOAP extensions that can ensure the integrity and confidentiality of messages. It accommodates a variety of security models and encryption technologies and is extensible to support multiple security token formats.

XA XA is an X/Open specification for distributed transaction processing. It describes the responsibilities of a resource manager for transactional processing.

XML Encryption XML Encryption is a W3C specification that defines how to encrypt the content of an XML element.

XML Schema XML schema is a W3C specification that describes how one can create an XML schema definition file (extension ".xsd"). The schema file is used for XML validation and defines rules to which an XML file document should conform.

XML Signature XML Signature is a W3C recommendation that defines an XML syntax for digital signatures.

XML validation Verifies that an XML file is syntactically well-formed and is conformant to a defined structure. The defined structure is typically specified using a DTD file or an XML schema.

XSLT Extensible Stylesheet Language Transformations (XSLT) is a language for the transformation of XML documents into another XML- or text-based document. XSLT is a W3C specification.

Appendix C Acronym List

API	application program interface
ATAM	Architecture Tradeoff Analysis Method
BPEL	Business Process Execution Language
CEP	complex event processing
CICS	Customer Information Control System
CIO	chief information officer
CORBA	Common Object Request Broker Architecture
COTS	commercial off-the-shelf
CSO	chief security officer
DCE	Distributed Computing Environment
DCOM	distributed component object model
DOM	document object model
DTD	document type definition
EAI	enterprise application integration
EDA	event-driven architecture
ERP	enterprise resource planning
ESB	enterprise service bus
ESP	event stream processing
ETL	extract, transform, and load
HIPAA	Health Insurance Portability and Accountability Act
IIOP	Internet Inter-ORB Protocol
J2EE	Java 2 Platform Enterprise Edition
JDBC	Java Database Connectivity
LDAP	Lightweight Directory Access Protocol
MVC	Model View Controller

OPC	Order Processing Center
ORB	object request broker
PKI	Public Key Infrastructure
RAS	reliability, availability, and serviceability
REST	Representational State Transfer
RIA	rich internet application
RPC	remote procedure call
SaaS	software as a service
SAML	Security Assertion Markup Language
SAX	simple API for XML
SCA	service component architecture
SDO	Service Data Objects
SEI	Software Engineering Institute
SLA	service level agreement
SNA/LU	systems network architecture/logical unit
SOA	service-oriented architecture
SOAP	Simple Object Access Protocol/Service Oriented Architecture Protocol (no longer an acronym, now simply referred to as SOAP)
SQL	Structured Query Language
SSL	secure socket layer
SSO	single sign on
UDDI	universal description, discovery, and integration
UML	Unified Modeling Language
URI	uniform resource identifier
VPN	Virtual Private Networking
WCF	Windows Communication Foundation
WS	Web services
WSDL	Web Services Description Language

WS-I	Web Services-Interoperability
WS-R	Web Services-Reliability
WSRM	WS-Reliability standard
WS-RM	WS-Reliable Messaging standard
XML	Extensible Markup Language
XQuery	XML Query Language
XSLT	Extensible Stylesheet Language Transformation

References

[Barbacci 2003]
Barbacci, Mario; Clements, Paul; Lattanze, Anthony; Northrop, Linda; & Wood, William. *Using the Architecture Tradeoff Analysis Method (ATAM) to Evaluate the Software Architecture for a Product Line of Avionics Systems: A Case Study* (CMU/SEI-2003-TN-012, ADA418415). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tn012.html.

[Bass 2001]
Bass, Len; Klein, Mark; & Moreno, Gabriel. *Applicability of General Scenarios to the Architecture Tradeoff Analysis Method* (CMU/SEI-2001-TR-014, ADA396098). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. http://www.sei.cmu.edu/publications/documents/01.reports/01tr014.html.

[Bass 2003]
Bass, Len; Clements, Paul; & Kazman, Rick. *Software Architecture in Practice*, 2nd ed. Boston, MA: Addison-Wesley, 2003 (ISBN 0-321-15495-9).

[Clements 2002a]
Clements, Paul; Bachmann, Felix; Bass, Len; Garlan, David; Ivers, James; Little, Reed; Nord, Robert; & Stafford, Judith. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley, 2002 (ISBN 0-201-70372-6).

[Clements 2002b]
Clements, Paul; Kazman, Rick; & Klein, Mark. *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA: Addison-Wesley, 2002 (ISBN 0201-70482-X).

[Ewald 2002]
Ewald, Tim. *The Argument Against SOAP Encoding.* http://msdn2.microsoft.com/en-us/library/ms995710.aspx (2002).

[Fielding 2000]
Fielding, Roy. "Architectural Styles and the Design of Network-Based Software Architectures." PhD diss., University of California, Irvine, 2000.

[Jones 2001]
Jones, Lawrence G. & Lattanze, Anthony J. *Using the Architecture Tradeoff Analysis Method to Evaluate a Wargame Simulation System: A Case Study* (CMU/SEI-2001-TN-022, ADA399795). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. http://www.sei.cmu.edu/publications/documents/01.reports/01tn022.html.

[Kazman 2000]
Kazman, Rick; Klein, Mark; & Clements, Paul. *ATAM: Method for Architecture Evaluation* (CMU/SEI-2000-TR-004, ADA382629). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000.
http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html.

[Kiss 2004]
Kiss, Roman. *SoapMSMQ Transport*.
http://www.codeproject.com/cs/webservices/SoapMSMQ.asp (2004).

[Lipson 2006]
Lipson, Howard & Peterson, Gunnar. *Security Concepts, Challenges, and Design Considerations for Web Services Integration*.
https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/assembly/639.html (2006).

[Lublinsky 2007]
Lublinsky, Boris. "Versioning in SOA." *The Architecture Journal, Journal 11* (April 2007): 36-41. http://msdn2.microsoft.com/en-us/library/bb491124.aspx (2007).

[MacVittie 2006]
MacVittie, Lori. "Taking a REST from SOAP." *Network Computing 17*, 20 (October 5, 2006): 25-26.
http://www.networkcomputing.com/channels/enterpriseapps/showArticle.jhtml?articleID=193005691 (2006).

[McConnell 2001]
McConnell, Steve. "An Ounce of Prevention." *IEEE Software 18*, 3 (May 2001): 5-7.

[Microsoft 2007]
Microsoft Corporation. *What Is Windows Communication Foundation?*
http://msdn2.microsoft.com/en-us/library/ms731082.aspx. (2007)

[Mitchell 2005]
Mitchell, Benjamin. "Why WSE?" http://msdn2.microsoft.com/en-us/library/ms996935.aspx (February 2005).

[OASIS 2004a]
OASIS. *Web Services Reliable Messaging TC WS-Reliability 1.1: OASIS Standard, 15 November 2004*.
http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf (2004).

[OASIS 2004b]
OASIS. *Web Services Security: SOAP Message Security 1.0 (WS4 Security 2004)*.
http://xml.coverpages.org/WSS-SOAP-MessageSecurityV10-20040315.pdf (15 March 2004).

[OASIS 2005]
OASIS. *SAML V2.0 Executive Overview: Committee Draft 01, 12 April 2005*. http://www.oasis-open.org/committees/download.php/13525/sstc-saml-exec-overview-2.0-cd-01-2col.pdf (2005).

[OASIS 2006a]
OASIS. *Web Services Business Process Execution Language Version 2.0*: *OASIS Standard 11 April 2007*. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf (2007).

[OASIS 2006b]
OASIS. *Web Services Reliable Messaging (WS-ReliableMessaging): Committee draft 04, August 11, 2006*. http://docs.oasis-open.org/ws-rx/wsrm/200608/wsrm-1.1-spec-cd-04.pdf (2006).

[O'Brien 2005]
O'Brien, Liam; Bass, Len; & Merson, Paulo. *Quality Attributes and Service-Oriented Architectures* (CMU/SEI-2005-TN-014, ADA441830). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents/05.reports/05tn014.html.

[OSOA 2007]
Open Service Oriented Architecture (OSOA). *Open SOA Service Component Architecture Specifications: Draft Specifications*. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications (2007)

[Schmelzer 2002]
Schmelzer, Ronald. "Breaking XML to Optimize Performance." ZapThink, 24 October 2002. http://searchwebservices.techtarget.com/originalContent/0,289142,sid26_gci858888,00.html (2002).

[SDO 2006]
Service Data Objects (SDO). *Service Data Objects for Java Specification Version 2.1.0*, *November 2006.* http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf?version=1 (2006).

[SEI 2007]
Software Engineering Institute. *Software Architecture Glossary*. http://www.sei.cmu.edu/architecture/glossary.html (2007).

[Shah 2006]
Shah, Gautam. "Axis Meets MOM: Reliable Web Services with Apache Axis and MOM." JavaWorld, February 20, 2006. http://www.javaworld.com/javaworld/jw-02-2006/jw-0220-axis.html (2006).

[Shaw 1996]
Shaw, M. & Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall, 1996 (ISBN 0-131-82957-2).

[Shirasuna 2004]
Shirasuna, Satoshi; Slominski, Aleksander; Fang, Liang; & Gannon, Dennis. "Performance Comparison of Security Mechanisms for Grid Services," 360-364. *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (Grid '04)*. Pittsburgh, PA, November 2004. IEEE, 2004. http://www.extreme.indiana.edu/xgws/papers/sec-perf-short.pdf.

[Singh 2004]
Singh, Inderjeet. *Designing Web Services with the J2EE 1.4 Platform: JAX-RPC, SOAP, and XML Technologies*. Boston, MA: Addison-Wesley, 2004 (ISBN 0-321-20521-9).

[Sun 2007a]
Sun Microsystems. *Java BluePrints Program*. https://blueprints.dev.java.net/ (2007).

[Sun 2007b]
Sun Microsystems, Inc. *JSR 311:JAX-RS: The Java API for RESTful Web Services*. http://jcp.org/en/jsr/detail?id=311 (2007).

[Vinoski 2007]
Vinoski, Steve. "REST Eye for the SOA Guy." *IEEE Internet Computing 11*, 1 (January/February 2007): 82-84.

[WS-I 2007]
Web Services Interoperability Organization (WS-I). *Basic Security Profile Version 1.0: Final Material 2007-03-30*. http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html (2007).

[W3C 2003]
W3C. *SOAP Version 1.2 Part 0: Primer*, *W3C Recommendation 24 June 2003*. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/ (2003).

[W3C 2007]
W3C. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*, *W3C Working Draft 26 March 2007*. http://www.w3.org/TR/wsdl20-primer/ (2007).

[xFront 2007]
xFront. *XML Schema Versioning*. http://www.xfront.com/Versioning.pdf (2007).

[Zimmermann 2003]
Zimmermann, Olaf; Tomlinson, Mark; & Peuser, Stefan. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects*. London, UK: Springer, 2003 (ISBN 978-3-540-00914-6).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2007	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Evaluating a Service-Oriented Architecture	FA8721-05-C-0003

6. AUTHOR(S)

Phil Bianco, Rick Kotermanski, & Paulo Merson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2007-TR-015

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2007-015

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

The emergence of service-oriented architecture (SOA) as an approach for integrating applications that expose services presents many new challenges to organizations resulting in significant risks to their business. Particularly important among those risks are failures to effectively address quality attribute requirements such as performance, availability, security, and modifiability. Because the risk and impact of SOA are distributed and pervasive across applications, it is critical to perform an architecture evaluation early in the software life cycle. This report contains technical information about SOA design considerations and tradeoffs that can help the architecture evaluator to identify and mitigate risks in a timely and effective manner. The report provides an overview of SOA, outlines key architecture approaches and their effect on quality attributes, establishes an organized collection of design-related questions that an architecture evaluator may use to analyze the ability of the architecture to meet quality requirements, and provides a brief sample evaluation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
service-oriented architecture, SOA, Web services, software architecture evaluation, ATAM, software architecture, design decisions, quality attributes	90

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL